



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

NW.js Essentials

Build native desktop applications for Windows, Mac OS, or Linux
using the latest web technologies

Alessandro Benoit

[PACKT] open source*
PUBLISHING community experience distilled

NW.js Essentials

Build native desktop applications for Windows, Mac OS,
or Linux using the latest web technologies

Alessandro Benoit



BIRMINGHAM - MUMBAI

NW.js Essentials

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2015

Production reference: 1190515

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78528-086-3

www.packtpub.com

Credits

Author

Alessandro Benoit

Project Coordinator

Harshal Ved

Reviewers

Dan Bendell

Marco Fabbri

Julio Freitas

Proofreaders

Stephen Copestake

Safis Editing

Indexer

Mariammal Chettiyar

Commissioning Editor

Amarabha Banerjee

Graphics

Disha Haria

Abhinash Sahu

Acquisition Editor

Reshma Raman

Content Development Editor

Gaurav Sharma

Production Coordinator

Alwin Roy

Technical Editor

Humera Shaikh

Cover Work

Alwin Roy

Copy Editor

Sarang Chari

About the Author

Alessandro Benoit is a 31-year-old web developer from Italy. He currently works both with backend and frontend technologies, ranging from PHP development, mostly on WordPress and Laravel, to web design and building open source jQuery plugins. He's also an early adopter of NW.js and the developer of Nuwk!, an open source application that simplifies the building process of NW.js applications on Mac OS X.

Acknowledgments

NW.js Essentials is my first attempt at becoming a technical book writer. Writing it has been fun and exciting but also quite harsh. There were moments between work and personal issues when I thought I couldn't make it. But here we are! What I want to do now is thank all the wonderful people who have accompanied me on this journey.

First of all, I want to thank the whole open source community. These people are the reason I'm in information technology in the first place. I can still remember when I was 14 and some guy on IRC was teaching me about breaking NetBios to gain access to remote hosts of Microsoft Windows (I swear, I've never done any harm); that was the first time I realized how thrilling technology was to me.

I really can't help being grateful to Roger Wang for all the effort he's taken to make NW.js an easy and stable environment to bring web technologies to our desktops.

I want to personally thank Reshma Raman and Gaurav Sharma, my editors at Packt Publishing. I have never met them in person, but still they have been positive and supportive during the whole writing process.

Thanks to Marco Fabbri, Dan Bendell, and Julio Freitas, for the great job they have done in reviewing this book.

I have to thank Abramo Capozzolo, my employer at Comodolab, for being so open to the use of the latest technologies and for putting people and research above revenues. And of course, I would like to thank my colleague and friend, Christian Pucci, for bringing me in, in the first place.

Thanks to my life partner, Elisa Rocchi, for being present and always understanding despite my recent mood swings. And thanks to my best friend, Andrea Valli, for still being my friend after months of absence.

Thanks to my mother, Patrizia Capacci, for loving me unconditionally.

Finally, I want to dedicate the publication of this book to an old friend who unfortunately is no longer here to share this joy. This is also for you, Andrea Benvenuti.

About the Reviewers

Dan Bendell is a budding young developer currently at the University of Plymouth studying computing and game development. He is set to finish his studies in 2016 after completing a year of work in the industry. Upon finishing his academic endeavors, he wishes to pursue his dream of either creating a start-up or working within a game company that will allow him to create games that truly engage his audience and work with a variety of new people on a daily basis.

Marco Fabbri is an experienced software engineer and former professor of distributed systems (middleware for multi-agent systems, ReST architecture, and Node.js) at the University of Bologna, Italy. He's also an active contributor in NW.js codebase.

Julio Freitas completed his graduation in computer science with specialization in information systems and technology. He has been a developer of web applications since the year 2000. He worked as a developer and Unix systems administrator in projects of grid computing with Java and PHP for 5 years at the Center for Weather Forecasting and Climate Studies/National Institute for Space Research (CPTEC/INPE), Brazil. He currently resides in England, where he works in a web systems company, and he is now creating his own start-up and acting as a full-stack web developer in projects focused on API development and security and applications for mobile devices using the MEAN stack and Ionic.

Julio has also reviewed *JavaScript Regular Expressions*, Loiane Groner, Packt Publishing.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	v
Chapter 1: Meet NW.js	1
NW.js under the hood	3
Features and drawbacks of NW.js	4
NW.js – usage scenarios	5
Popular NW.js applications	5
Downloading and installing NW.js	6
Installing NW.js on Mac OS X	6
Installing NW.js on Microsoft Windows	7
Installing NW.js on Linux	8
Development tools	9
Writing and running your first "Hello World" app	10
Running NW.js applications on Sublime Text 2	12
Running NW.js applications on Microsoft Windows	12
Running NW.js applications on Mac OS	13
Running NW.js applications on Linux	13
Summary	14
Chapter 2: NW.js Native UI APIs	15
The App API – the core of your applications	17
Opening a file in your application natively	18
Accessing the application data folder path	19
Accessing the manifest file data	20
Best practices for closing applications	21
Registering system-wide hotkeys	22
Other app APIs	23
The Window API – working with windows on NW.js	24
Instantiating a new window object	24
Window – setting size and position of windows	26

Changing the window status	28
Fullscreen windows and the Kiosk mode	29
Frameless windows and drag regions	31
The taskbar icon – get the user's attention!	32
Closing windows	33
Other Window APIs	34
The Screen API – screen geometry functions	36
The Menu API – handling window and context menus	37
The contextual menu	38
The window menu	40
File dialogs – opening and saving files	41
Opening multiple files	43
Filtering by file type	43
Opening a directory	43
Saving files	43
Suggesting a default path	43
Opening files through file dragging	44
The Tray API – hide your application in plain sight	44
The Clipboard API – accessing the system clipboard	47
The Shell API – platform-dependent desktop functions	48
Summary	48
Chapter 3: Leveraging the Power of Node.js	49
Routing and templating in NW.js	50
Node.js global and process objects	51
The window object	53
Using NW.js' main module	54
Handling paths in NW.js	55
NW.js context issues	57
Working with Node.js modules	59
Internal modules	59
Third-party modules written in JavaScript	60
Third-party modules with C/C++ add-ons	60
Summary	61
Chapter 4: Data Persistence Solutions and Other	
Browser Web APIs	63
Data persistence solutions	64
Web storage	64
Web SQL Database	68
IndexedDB	72

XMLHttpRequest and BLOBs	79
Handling media files	80
Shedding some light on security issues	81
The Web Notifications API	83
Summary	85
Chapter 5: Let's Put It All Together	87
Let's get started!	88
A matter of style	90
The HTML5 skeleton	93
Let's dive deep into the application logic	94
The application layer	96
Adding a new task	98
Loading all the tasks	102
Implementing export and sync features	103
The NativeUI layer	105
Implementing the Window menu	106
Implementing the Context menu	111
Restoring the window position	111
Implementing the Options window	112
Closing the application	116
Making the application open smoothly	117
Summary	117
Chapter 6: Packaging Your Application for Distribution	119
The manifest file	120
The general logic behind the packaging procedure	123
Packaging NW.js applications for Mac OS X	125
Associating a file extension with your application	127
Packaging NW.js applications for Microsoft Windows	128
Registering a file type association on Microsoft Windows	130
Packaging NW.js applications for Linux	130
Adding icon and file type associations on Linux	131
Securing your source code	132
About NW.js application licensing	134
Summary	134
Chapter 7: Automated Packaging Tools	135
Web2Executable	135
node-webkit-builder and grunt-node-webkit-builder	137
grunt-node-webkit-builder	144
generator-node-webkit	145
Summary	149

Chapter 8: Let's Debug Your Application	151
Remote debugging	153
The DevTools API	154
Live reloading NW.js	156
Troubleshooting common issues	156
Summary	157
Chapter 9: Taking Your Application to the Next Level	159
NW.js boilerplates	159
node-webkit-hipster-seed	159
angular-desktop-app	160
node-webkit-tomster-seed	160
node-webkit-boilerplate	160
nw-boilerplate	160
Development ideas	160
Resources and tutorials	162
Summary	162
Index	165

Preface

NW.js is a web app runtime, based on Node.js and the Chromium open source browser project, which enables web developers to build native-like desktop applications for Windows, Mac OS X, or Linux, leveraging all the power of well-known web technologies such as Node.js, HTML5, and CSS.

In *NW.js Essentials*, you'll be guided through the full development process, starting from the theoretical basis behind NW.js technology to the realization of a fully working, multiplatform desktop application.

What this book covers

Chapter 1, Meet NW.js, provides a presentation of the NW.js technology followed by a brief digression on how NW.js works under the hood, an analysis of the strengths of the library, and eventually a step-by-step tutorial on building a first, simple "Hello World" application.

Chapter 2, NW.js Native UI APIs, takes you through a thorough description of Native UI APIs with examples on how to interact with OS windows, work with menus, run shell commands, and much, much more.

Chapter 3, Leveraging the Power of Node.js, gives an introduction of how NW.js development differs from standard server/client programming and a few interesting hints on how to get the maximum out of Node.js within NW.js applications.

Chapter 4, Data Persistence Solutions and Other Browser Web APIs, explains more about data handling and takes advantage of Browser Web APIs to build beautiful, usable, and native-like desktop applications.

Chapter 5, Let's Put It All Together, builds a fully working application based on many of the concepts you've learned in the previous chapters.

Chapter 6, Packaging Your Application for Distribution, gives a step-by-step tutorial on the packaging process of NW.js applications on Microsoft Windows, Mac OS X, and Linux.

Chapter 7, Automated Packaging Tools, takes advantage of third-party tools to simplify the packaging process of NW.js applications.

Chapter 8, Let's Debug Your Application, teaches you different ways to debug your application and a few common issues of NW.js.

Chapter 9, Taking Your Application to the Next Level, involves a collection of ideas and resources to get the best out of what you've learned in this book.

What you need for this book

In order to fully understand the concepts explained in the book, a decent knowledge of the following subjects is mandatory:

- Node.js programming and working with Node.js modules
- Modern web application languages such as HTML5, CSS3, and JavaScript
- Being comfortable with the use of the Unix terminal and the Microsoft Windows command line

It's not mandatory, but it might be helpful to have previous experience in the use of *JavaScript task runners* such as Gulp and Grunt.

Who this book is for

The book is targeted at experienced Node.js developers with a basic understanding of frontend web development.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Packages for Mac and Windows are zipped, while those for Linux are in the `tar.gz` format."

A block of code is set as follows:

```
{
  "name": "nw-hello-world",
  "main": "index.html",
  "dependencies": {
    "markdown": "0.5.x"
  }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:


```
var markdown = require("markdown").markdown,
    div = document.createElement("div"),
    content = "#Hello World!\n" +
    "We are using **io.js** " +
    "version * + process.version + ***;"


div.innerHTML = markdown.toHTML(content);
document.body.appendChild(div);
```

Any command-line input or output is written as follows:

```
$ cd nwjs
$ ./nw
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Open it, and from the top menu, navigate to **Tools** | **Build System** | **New Build System**."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Meet NW.js

Until a while ago, developing a desktop application that was compatible with the most common operating systems required an enormous amount of expertise, different programming languages, and logics for each platform.

Yet, for a while now, the evolution of web technologies has brought to our browsers many web applications that have nothing to envy from their desktop alternative. Just think of Google apps such as Gmail and Calendar, which, for many, have definitely replaced the need for a local mail client. All of this has been made possible thanks to the amazing potential of the latest implementations of the Browser Web API combined with the incredible flexibility and speed of the latest server technologies.

Although we live in a world increasingly interconnected and dependent on the Internet, there is still the need for developing desktop applications for a number of reasons:

- To overcome the lack of vertical applications based on web technologies
- To implement software solutions where data security is essential and cannot be compromised by exposing data on the Internet
- To make up for any lack of connectivity, even temporary
- Simply because operating systems are still locally installed

Once it's established that we cannot completely get rid of desktop applications and that their implementation on different platforms requires an often prohibitive learning curve, it comes naturally to ask: why not make desktop applications out of the very same technologies used in web development?

The answer, or at least one of the answers, is **NW.js**!

NW.js doesn't need any introduction. With *more than 20,000 stars on GitHub* (in the top four hottest C++ projects of the repository-hosting service) NW.js is definitely *one of the most promising projects* to create desktop applications with web technologies. Paraphrasing the description on GitHub, NW.js is a **web app runtime** that allows the browser DOM to access Node.js modules directly.

Node.js is responsible for hardware and operating system interaction, while the browser serves the graphic interface and implements all the functionalities typical of web applications. Clearly, the use of the two technologies may overlap; for example, if we were to make an asynchronous call to the API of an online service, we could use either a Node.js HTTP client or an XMLHttpRequest Ajax call inside the browser.

Without going into technical details, in order to create desktop applications with NW.js, all you need is a decent understanding of Node.js and some expertise in developing HTML5 web apps.

In this first chapter, we are going to dissect the topic dwelling on these points:

- A brief technical digression on how NW.js works
- An analysis of the pros and cons in order to determine use scenarios
- Downloading and installing NW.js
- Development tools
- Making your first, simple "Hello World" application



Important notes about NW.js (also known as Node-Webkit) and io.js

Before January 2015, since the project was born, NW.js was known as **Node-Webkit**. Moreover, with Node.js getting a little sluggish, much to the concern of V8 JavaScript engine updates, from version 0.12.0, NW.js is not based on Node.js but on **io.js**, an npm-compatible platform originally based on Node.js. For the sake of simplicity in the book, we will keep referring to Node.js even when talking about io.js as long as this does not affect a proper comprehension of the subject.

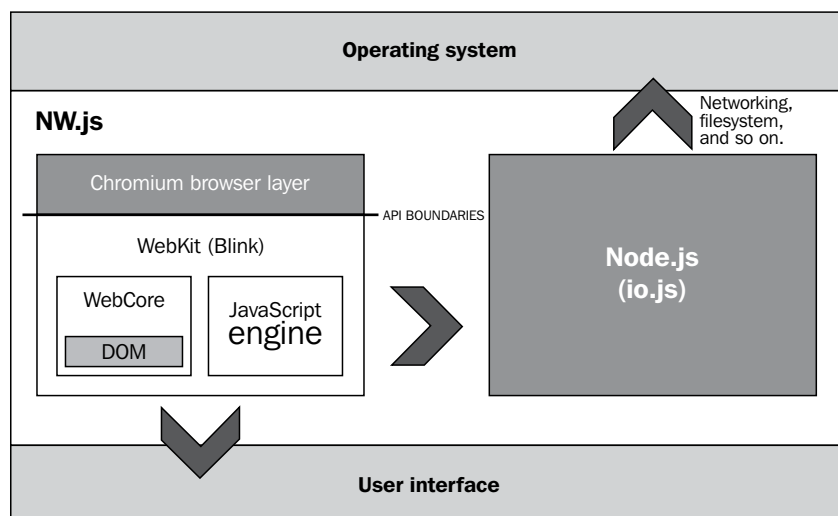
NW.js under the hood

As we stated in the introduction, NW.js, made by **Roger Wang** of *Intel's Open Source Technology Center* (Shanghai office) in 2011, is a web app runtime based on Node.js and the Chromium open source browser project. To understand how it works, we must first analyze its two components:

- Node.js is an efficient JavaScript runtime written in C++ and based on the V8 JavaScript engine developed by Google. Residing in the operating system's application layer, Node.js can access hardware, filesystems, and networking functionalities, enabling its use in a wide range of fields, from the implementation of web servers to the creation of control software for robots. (As we stated in the introduction, NW.js has replaced Node.js with io.js from version 0.12.0.)
- WebKit is a layout engine that allows the rendering of web pages starting from the DOM, a tree of objects representing the web page. NW.js is actually not directly based on WebKit but on **Blink**, a fork of WebKit developed specifically for the Chromium open source browser project and based on the V8 JavaScript engine as is the case with Node.js.

Since the browser, for security reasons, cannot access the application layer and since Node.js lacks a graphical interface, Roger Wang had the insight of combining the two technologies by creating NW.js.

The following is a simple diagram that shows how Node.js has been combined with WebKit in order to give NW.js applications access to both the GUI and the operating system:



In order to integrate the two systems, which, despite speaking the same language, are very different, a couple of tricks have been adopted. In the first place, since they are both event-driven (following a logic of action/reaction rather than a stream of operations), the event processing has been unified. Secondly, the Node context was injected into WebKit so that it can access it.

The amazing thing about it is that you'll be able to program all of your applications' logic in JavaScript with no concerns about where Node.js ends and WebKit begins.

Today, NW.js has reached version 0.12.0 and, although still young, is one of the most promising web app runtimes to develop desktop applications adopting web technologies.

Features and drawbacks of NW.js

Let's check some of the features that characterize NW.js:

- NW.js allows us to realize modern desktop applications using HTML5, CSS3, JS, WebGL, and the full potential of Node.js, including the use of third-party modules
- The Native UI API allows you to implement native lookalike applications with the support of menus, clipboards, tray icons, and file binding
- Since Node.js and WebKit run within the same thread, NW.js has excellent performance
- With NW.js, it is incredibly easy to port existing web applications to desktop applications
- Thanks to the CLI and the presence of third-party tools, it's really easy to debug, package, and deploy applications on Microsoft Windows, Mac OS, and Linux

However, all that glitters is not gold. There are some cons to consider when developing an application with NW.js:

- **Size of the application:** Since a copy of NW.js (70-90 MB) must be distributed along with each application, the size of the application makes it quite expensive compared to native applications. Anyway, if you're concerned about download times, compressing NW.js for distribution will save you about half the size.
- **Difficulties in distributing your application through Mac App Store:** In this book, it will not be discussed (just do a search on Google), but even if the procedure is rather complex, you can distribute your NW.js application through Mac App Store. At the moment, it is not possible to deploy a NW.js application on Windows Store due to the different architecture of .appx applications.

- **Missing support for iOS or Android:** Unlike other SDKs and libraries, at the moment, it is not possible to deploy an NW.js application on iOS or Android, and it does not seem to be possible to do so in the near future. However, the *portability* of the HTML, JavaScript, and CSS code that can be distributed on other platforms with tools such as PhoneGap or TideSDK should be considered. Unfortunately, this is not true for all of the features implemented using Node.js.
- **Stability:** Finally, the platform is still quite young and not bug-free.

NW.js – usage scenarios

The flexibility and good performance of NW.js allows its use in countless scenarios, but, for convenience, I'm going to report only a few notable ones:

- Development tools
- Implementation of the GUI around existing CLI tools
- Multimedia applications
- Web services clients
- Video games

The choice of development platform for a new project clearly depends only on the developer; for the overall aim of confronting facts, it may be useful to consider some specific scenarios where the use of NW.js might not be recommended:

- When developing for a specific platform, graphic coherence is essential, and, perhaps, it is necessary to distribute the application through a store
- If the performance factor limits the use of the preceding technologies
- If the application does a massive use of the features provided by the application layer via Node.js and it has to be distributed to mobile devices

Popular NW.js applications

After summarizing the pros and cons of NW.js, let's not forget the real strength of the platform – the many applications built on top of NW.js that have already been distributed. We list a few that are worth noting:


- **Wunderlist for Windows 7:** This is a to-do list / schedule management app used by millions
- **Cellist:** This is an HTTP debugging proxy available on Mac App Store

- **Game Dev Tycoon:** This is one of the first NW.js games that puts you in the shoes of a 1980s game developer
- **Intel® XDK:** This is an HTML5 cross-platform solution that enables developers to write web and hybrid apps

Downloading and installing NW.js

Installing NW.js is pretty simple, but there are many ways to do it. One of the easiest ways is probably to run `npm install nw` from your terminal, but for the educational purposes of the book, we're going to manually download and install it in order to properly understand how it works.

You can find all the download links on the project website at <http://nwjs.io/> or in the **Downloads** section on the GitHub project page at <https://github.com/nwjs/nw.js/>; from here, download the package that fits your operating system.

[ For example, as I'm writing this book, Node-Webkit is at version 0.12.0, and my operating system is Mac OS X Yosemite 10.10 running on a 64-bit MacBook Pro; so, I'm going to download the `nwjs-v0.12.0-osx-x64.zip` file.]

Packages for Mac and Windows are zipped, while those for Linux are in the `tar.gz` format. Decompress the files and proceed, depending on your operating system, as follows.

Installing NW.js on Mac OS X

Inside the archive, we're going to find three files:

- `Credits.html`: This contains credits and licenses of all the dependencies of NW.js
- `nwjs.app`: This is the actual NW.js executable
- `nwjc`: This is a CLI tool used to compile your source code in order to protect it

[ Before v0.12.0, the filename of `nwjc` was `nwsnapshot`.]

Currently, the only file that interests us is `nwjs.app` (the extension might not be displayed depending on the OS configuration). All we have to do is copy this file in the `/Applications` folder—your main applications folder.



If you'd rather install NW.js using **Homebrew Cask**, you can simply enter the following command in your terminal:

```
$ brew cask install nw
```

If you are using Homebrew Cask to install NW.js, keep in mind that the Cask repository might not be updated and that the `nwjs.app` file will be copied in `~/Applications`, while a symlink will be created in the `/Applications` folder.

Installing NW.js on Microsoft Windows

Inside the Microsoft Windows NW.js package, we will find the following files:

- `credits.html`: This contains the credits and licenses of all NW.js dependencies
- `d3dcompiler_47.dll`: This is the Direct3D library
- `ffmpegsumo.dll`: This is a media library to be included in order to use the `<video>` and `<audio>` tags
- `icudtl.dat`: This is an important network library
- `libEGL.dll`: This is the WebGL and GPU acceleration
- `libGLESv2.dll`: This is the WebGL and GPU acceleration
- `locales/`: This is the languages folder
- `nw.exe`: This is the actual NW.js executable
- `nw.pak`: This is an important JS library
- `pdf.dll`: This library is used by the web engine for printing
- `nwjc.exe`: This is a CLI tool to compile your source code in order to protect it

Some of the files in the folder will be omitted during the final distribution of our application, but for development purposes, we are simply going to copy the whole content of the folder to `C:/Tools/nwjs`.

Installing NW.js on Linux

On Linux, the procedure can be more complex depending on the distribution you use. First, copy the downloaded archive into your home folder if you have not already done so, and then open the terminal and type the following command to unpack the archive (change the version accordingly to the one downloaded):

```
$ gzip -dc nwjs-v0.12.0-linux-x64.tar.gz | tar xf -
```

Now, rename the newly created folder in `nwjs` with the following command:

```
$ mv ~/nwjs-v0.12.0-linux-x64 ~/nwjs
```

Inside the `nwjs` folder, we will find the following files:

- `credits.html`: This contains the credits and licenses of all the dependencies of NW.js
- `icudtl.dat`: This is an important network library
- `libffmpegsumo.so`: This is a media library to be included in order to use the `<video>` and `<audio>` tags
- `locales/`: This is a languages folder
- `nw`: This is the actual NW.js executable
- `nw.pak`: This is an important JS library
- `nwjc`: This is a CLI tool to compile your source code in order to protect it

Open the folder inside the terminal and try to run NW.js by typing the following:

```
$ cd nwjs
$ ./nw
```

If you get the following error, you are probably using a version of **Ubuntu** later than 13.04, **Fedora** later than 18, or another Linux distribution that uses `libudev.so.1` instead of `libudev.so.0`: otherwise, you're good to go to the next step:

```
error while loading shared libraries: libudev.so.0: cannot open shared
object file: No such file or directory
```

Until NW.js is updated to support `libudev.so.1`, there are several solutions to solve the problem. For me, the easiest solution is to type the following terminal command inside the directory containing `nw`:

```
$ sed -i 's/udev\.so\.0/udev.so.1/g' nw
```

This will replace the string related to `libudev`, within the application code, with the new version. The process may take a while, so wait for the terminal to return the cursor before attempting to enter the following:

```
$ ./nw
```

Eventually, the NW.js window should open properly.

Development tools

As you'll make use of third-party modules of Node.js, you're going to need **npm** in order to download and install all the dependencies; so, Node.js (<http://nodejs.org/>) or io.js (<https://iojs.org/>) must be obviously installed in your development environment.

I know you cannot wait to write your first application, but before you start, I would like to introduce you to **Sublime Text 2**. It is a simple but sophisticated *IDE*, which, thanks to the support for custom build scripts, allows you to run (and debug) NW.js applications from inside the editor itself.

If I wasn't convincing and you'd rather keep using your favorite IDE, you can skip to the next section; otherwise, follow these steps to install and configure Sublime Text 2:

1. Download and install Sublime Text 2 for your platform from <http://www.sublimetext.com/>.
2. Open it and from the top menu, navigate to **Tools | Build System | New Build System**.
3. A new edit screen will open; paste the following code depending on your platform:

- On Mac OS X:

```
{
  "cmd": ["nwjs", "--enable-logging",
    "${project_path}:${file_path}"],
  "working_dir": "${project_path}:${file_path}",
  "path": "/Applications/nwjs.app/Contents/MacOS/"
}
```

- On Microsoft Windows:

```
{
  "cmd": ["nw.exe", "--enable-logging",
    "${project_path}:${file_path}"],
  "working_dir": "${project_path}:${file_path}",
  "path": "C:/Tools/nwjs/",
}
```


```
"shell": true
}

◦ On Linux:

{
  "cmd": ["nw", "--enable-logging",
    "${project_path:${file_path}}"],
  "working_dir": "${project_path:${file_path}}",
  "path": "/home/username/nwjs/"
}
```

4. Type *Ctrl* + *S* (*Cmd* + *S* on Mac) and save the file as `nw-js.sublime-build`.

Perfect! Now you are ready to run your applications directly from the IDE.

 There are a lot of packages, such as *SublimeLinter*, *LiveReload*, and *Node.js code completion*, available to Sublime Text 2. In order to install them, you have to install **Package Control** first. Just open <https://sublime.wbond.net/installation> and follow the instructions.

Writing and running your first "Hello World" app

Finally, we are ready to write our first simple application. We're going to revisit the usual "Hello World" application by making use of a Node.js module for markdown parsing.

"Markdown is a plain text formatting syntax designed so that it can be converted to HTML and many other formats using a tool by the same name."

– Wikipedia

Let's create a `Hello World` folder and open it in Sublime Text 2 or in your favorite IDE. Now open a new `package.json` file and type in the following JSON code:

```
{
  "name": "nw-hello-world",
  "main": "index.html",
  "dependencies": {
    "markdown": "0.5.x"
  }
}
```



The `package.json` **manifest file**, as you'll see in *Chapter 6, Packaging Your Application for Distribution*, is essential for distribution as it determines many of the window properties and primary information about the application. Moreover, during the development process, you'll be able to declare all of the dependencies.

In this specific case, we are going to assign the application name, the main file, and obviously our dependency, the **markdown module**, written by *Dominic Baggott*.



If you so wish, you can create the `package.json` manifest file using the `npm init` command from the terminal as you're probably used to already when creating npm packages.

Once you've saved the `package.json` file, create an `index.html` file that will be used as the main application file and type in the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <script>
      <!--Here goes your code-->
    </script>
  </body>
</html>
```

As you can see, it's a very common *HTML5 boilerplate*. Inside the script tag, let's add the following:

```
var markdown = require("markdown").markdown,
    div = document.createElement("div"),
    content = "#Hello World!\n" +
      "We are using io.js " +
      "version * + process.version + *";

div.innerHTML = markdown.toHTML(content);
document.body.appendChild(div);
```

What we do here is `require` the `markdown` module and then parse the `content` variable through it. To keep it as simple as possible, I've been using *Vanilla JavaScript* to output the parsed HTML to the screen. In the highlighted line of code, you may have noticed that we are using `process.version`, a property that is a part of the Node.js context.



If you try to open `index.html` in a browser, you'd get the **Reference Error: require is not defined** error as Node.js has not been injected into the WebKit process.

Once you have saved the `index.html` file, all that is left is to install the dependencies by running the following command from the terminal inside the project folder:

```
$ npm install
```

And we are ready to run our first application!

Running NW.js applications on Sublime Text 2

If you opted for Sublime Text 2 and followed the procedure in the development tools section, simply navigate to **Project | Save Project As** and save the `hello-world.sublime-project` file inside the project folder.

Now, in the top menu, navigate to **Tools | Build System** and select **nw.js**. Finally, press `Ctrl + B` (or `Cmd + B` on Mac) to run the program.

If you have opted for a different IDE, just follow the upcoming steps depending on your operating system.

Running NW.js applications on Microsoft Windows

Open the command prompt and type:

```
C:\> c:\Tools\nwjs\nw.exe c:\path\to\the\project\
```



On Microsoft Windows, you can also drag the folder containing `package.json` to `nw.exe` in order to open it.

Running NW.js applications on Mac OS

Open the terminal and type:

```
$ /Applications/nwjs.app/Contents/MacOS/nwjs /path/to/the/project/
```

Or, if running NW.js applications inside the directory containing `package.json`, type:

```
$ /Applications/nwjs.app/Contents/MacOS/nwjs .
```



As you can see in Mac OS X, the NW.js kit's executable binary is in a hidden directory within the `.app` file.

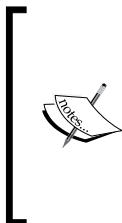
Running NW.js applications on Linux

Open the terminal and type:

```
$ ~/nwjs/nw /path/to/the/project/
```

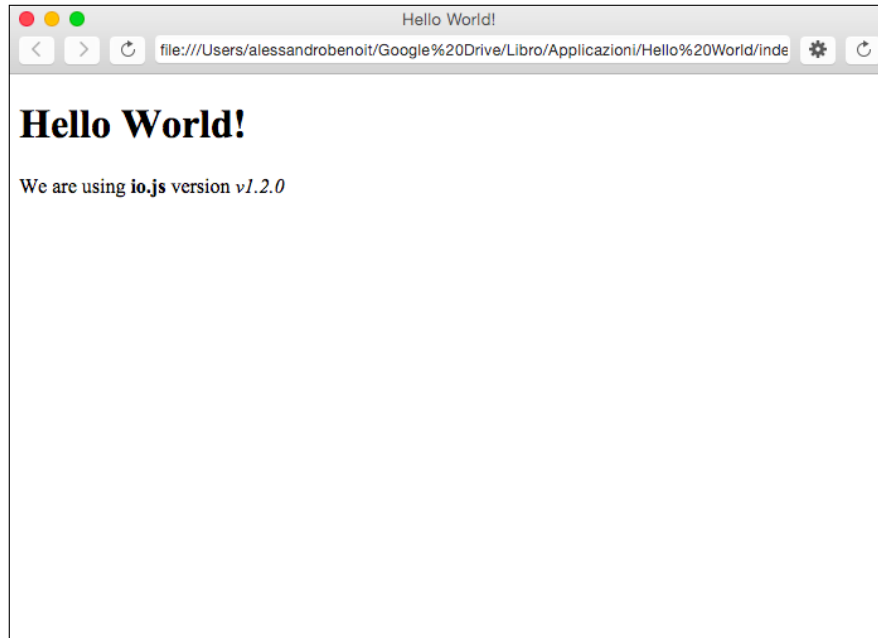
Or, if running NW.js applications inside the directory containing `package.json`, type:

```
$ ~/nwjs/nw .
```



Running the application, you may notice that a few errors are thrown depending on your platform. As I stated in the pros and cons section, NW.js is still young, so that's quite normal, and probably we're talking about minor issues. However, you can search in the NW.js GitHub issues page in order to check whether they've already been reported; otherwise, open a new issue—your help would be much appreciated.

Now, regardless of the operating system, a window similar to the following one should appear:



As illustrated, the `process.version` object variable has been printed properly as Node.js has correctly been injected and can be accessed from the DOM.

Perhaps, the result is a little different than what you expected since the *top navigation bar* of Chromium is visible. Do not worry! You can get rid of the navigation bar at any time simply by adding the `window.toolbar = false` parameter to the manifest file, but for now, it's important that the bar is visible in order to debug the application.

Summary

In this chapter, you discovered how NW.js works under the hood, the recommended tools for development, a few usage scenarios of the library, and eventually, how to run your first, simple application using third-party modules of Node.js. I really hope I haven't bored you too much with the theoretical concepts underlying the functioning of NW.js; I really did my best to keep it short.

In the following chapter, we will be much more operational and discover how to interact with the operating system thanks to the **Native UI APIs** of NW.js. In *Chapter 2, NW.js Native UI APIs*, you will indeed discover how to handle *multiple windows*, work with *file dialogs*, create *custom menus*, and much, much more!

2

NW.js Native UI APIs

As we have seen in the first chapter, it is incredibly easy to build desktop applications out of a simple HTML page but, in order to really understand the power of NW.js, we have to push it a step further. We know that **Node.js** takes care of dealing with low-level system functionality while **WebKit** handles the window GUI and all the stuff you'd usually do in a web application, but there is still something missing.

In order to *create a decent user experience*, we need to give our users the feeling that they are dealing with a **native desktop application** with the ability to handle multiple windows, access the clipboard, or hide in the system tray. To sum up, our application needs to be fully integrated with the OS GUI.

This cannot be done by WebKit because of the *API boundaries*, nor can it be done by Node.js as it doesn't have access to the GUI, but fortunately, NW.js gives us the full thing, adding on top of the others a new layer of JavaScript APIs that can interact with the OS graphic interface.

We're talking about **NW.js Native UI APIs**.

Native UI APIs are a custom set of APIs that let you create *native UI controls* in NW.js. In order to use them, you first have to require `nw.gui` as follows:

```
var gui = require('nw.gui');
```

The `gui` object is nothing more than a library containing a reference to all the APIs we're going to deal with in this chapter. So, for example, if you'd need to close the current window, you'd code something like the following:

```
var gui = require('nw.gui');
var currentWindow = gui.Window.get();
currentWindow.close();
```



To keep it as simple as possible, all the examples reported in this chapter are to be considered as inline and hence to be included between the script or body tags inside a pretty basic HTML5 boilerplate. In order to run them, follow the procedure described in *Chapter 1, Meet NW.js*, for the Hello World application. One more thing to consider is that in a few examples, I'm using `console.log()` in order to display data. We haven't talked about *debugging* yet, but for the moment, all you need to know is that when the toolbar is visible, clicking on the top-left icon near the refresh button will show the **DevTools** window.

As illustrated, we're requiring the `nw.gui` library, instantiating a window object for the current window, and then, eventually, calling the `close` method on it.

One more thing to consider about Native UI APIs is that each object inherits from **Node.js' EventEmitter** in order to listen and aptly respond to generated events.

Let's see an example:

```
var gui = require('nw.gui');
var currentWindow = gui.Window.get();

currentWindow.on('minimize', function() {
  alert('Minimizing the window');
});
```

This is the same as before, but this time, instead of calling a method, we're waiting for the `minimize` event to be fired in order to trigger an alert. We're going to go deeper into the **Window API** subject in the coming sections, but, for now, this is pretty much all you need to know about it.

There's one more thing to consider when using Native UI APIs; if you're doing something wrong, the app will crash—no exceptions thrown! That's why, you should *follow these good practices*:

- Remember to assign a `null` value to deleted/removed elements in order to avoid misbehaviors in case you reuse them
- When possible, do not recreate UI elements—reuse them
- Do not reassign elements
- Do not change the UI type's prototype

Here's a short summary of all the APIs you're going to master in this chapter:

APIs	Description
App	This lets you interact with the basic functionality of the application, including interactions such as opening bound file types, accessing the manifest file, registering global key combinations, or quitting the application.
Window	These are the extended Window APIs that let you handle one or multiple windows, get the user attention, and respond to window events.
Screen	This is a singleton that lets you get screen information and respond to screen events, such as a resolution change or a display addition.
Menu	This enables us to create window, tray, and contextual menus.
File dialogs	This lets us open or save files through file dialogs.
Tray	This lets us enable and manage tray/status icons.
Clipboard	This lets us access the system clipboard.
Shell	This lets us open files and URIs with the OS default applications.



In this chapter, I'm going to illustrate only the APIs I consider essential for developing native look-alike applications. Since there are a few more, which are not consistent with the purpose of the book, I will reference them in a summary list at the end of each section. You can learn more about them on the GitHub wiki page of the project at <https://github.com/nwjs/nw.js/wiki/Native-UI-API-Manual>.

The App API – the core of your applications

Eventually, we can dive deep into the actual code behind NW.js applications. The app API provides a useful set of functionalities, which you can leverage to *access main information and events of your application*. We shall start describing, one by one, all the functionalities provided.

Opening a file in your application natively

There are many ways to open files in an NW.js application; in this section, we're going to deal with the opening of *binded file types*. Let's say you're developing a text editor; you'd probably want `.txt` files to be associated with it.

Actually, the real binding is made on the packaging step described in *Chapter 6, Packaging Your Application for Distribution*, but to open a file using the OS functionality (for example, through the **Open with** contextual menu), we need to first be able to read the arguments passed when the application gets started.

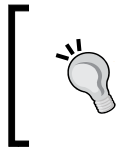
As a matter of fact, when you choose to open a given file with an application, here's what happens:

```
$ nw path/to/nwjs/app path/to/file.txt
```

You might even open more than one file at a time, as follows:

```
$ nw path/to/nwjs/app path/to/file.txt path/to/other_file.txt
```

In order to access these file paths, we need to access the `App.argv` property that indeed provides an array of arguments passed to the application executable.



In order to reproduce the next examples, you'll first need to package your application as described in *Chapter 6, Packaging Your Application for Distribution*, as you won't be able to pass arguments to your application while it is in development.

Here's a brief example showing how to intercept those arguments:

```
var gui = require('nw.gui');
var args = gui.App.argv;

// If arguments are passed
if (args.length > 0){
  // For each of them
  args.forEach(function (arg) {
    // Check if arg is valid file type and do something with it
  });
}
```

As illustrated, we intercept arguments and, if there are any, we loop through them in order to check whether they're of a valid file type and eventually do something with them.

The problem with this solution is that it is *run only once*, when the application is first started. As you probably don't want to run a new instance of the application for each opened file, you'll need to intercept the open event.

The open event is fired indeed only when an instance of the application has already been started. Check the following example:

```
var gui = require('nw.gui');
gui.App.on('open', function(cmdline) {
  console.log('command line: ' + cmdline);
});
```

Unlike `App.argv`, which stores all parameters in an array, the open event is fired once for each file you're passing as an argument. The callback parameter `cmdline` is indeed a string variable containing the file path.



When dragging files into the application icon, their path will be passed to the application in the same way that it would with **Open with**.

Accessing the application data folder path

All the operating systems we're dealing with provide a default folder, specific for each application and each user, to store personal settings, application-support files, and, in some cases, data. In order to avoid hardcoding this folder path for each platform, you can simply retrieve it using the `App.dataPath` property. Use the following code:

```
var gui = require('nw.gui');
alert(gui.App.dataPath);
```

The preceding code will pop up one of the following values depending on the operating system:

- **Microsoft Windows:** %LOCALAPPDATA%/<name>
- **Linux:** ~/.config/<name>
- **Mac OS X:** ~/Library/Application Support/<name>

In the preceding list of values, `<name>` is the application name as set in the manifest file. Let's see an example:

```
var fs = require('fs'),
    gui = require('nw.gui'),
    path = require('path');

var settings = {
  'show_sidebar' : true,
  'show_icons'   : false
};
var settingsFilename = path.join(gui.App.dataPath,
  'mySettings.json');

// Write settings file
fs.writeFile(settingsFilename, JSON.stringify(settings));

// Read settings file
fs.readFile(settingsFilename, 'utf8', function (err, data) {
  var savedSettings = JSON.parse(data);
  console.log(savedSettings);
});
```

In the preceding example, we're storing some settings in a JSON file inside the application data folder. As you can see, in the highlighted row, we're using `App.dataPath` to retrieve the path, so the code will work regardless of the platform.

Calling `fs.readFile` straight after `fs.writeFile` on the same file is obviously wrong as we have no guarantee that the file will be ready for reading at that point. In production, a callback should be used instead.

Accessing the manifest file data

As we've defined already, the `package.json` manifest file is the main configuration file of the application. Although we haven't yet examined it in the NW.js context (as we will in *Chapter 6, Packaging Your Application for Distribution*), that doesn't change the fact that sooner or later you're going to need the ability to read some data from it. Obviously, you could do it with Node.js, but it would be pointless as the manifest data is already stored in the `App.manifest` object. Here's a very basic example:

```
var gui = require('nw.gui');
var manifestData = gui.App.manifest;
alert(manifestData.name);
```

This will open an alert box containing the name of the application as set in the manifest file.

Best practices for closing applications

Before explaining how `App.closeAllWindows()` works, we should first understand an important thing about the NW.js application life cycle; by default, in order to quit an application, you first have to close all of its windows. Fortunately, we can easily change the default behavior.

Let's say, for example, we have an application with a main window and a settings window and we want to achieve the following result:

- Closing the main windows should close all the application windows and quit
- Closing the settings window should save the settings

Hence, when closing the main windows, the settings window should have an opportunity to save the data before quitting. Here's how we can implement that:

- `index.html` (main window):

```
var gui = require('nw.gui'),
    mainWindow = gui.Window.get();

window.open('settings.html');

mainWindow.on('close', function() {
    gui.App.closeAllWindows();
    this.close(true);
});
```
- `settings.html` (settings window):

```
var gui = require('nw.gui'),
    settingsWindow = gui.Window.get();

settingsWindow.on('close', function() {
    // Save data here
    this.close(true);
});
```


What's happening here is pretty cool. In `index.html`, we listen for the `close` event and send a closing signal to all the other windows with `App.closeAllWindows()`. In `settings.html`, we listen for the `close` event in order to save data before closing the window. Once the data is saved and all the windows are closed, the application can finally quit for good. As you guessed already, `App.closeAllWindows()` is very handy as it gives windows a *chance to save data before quitting*. Don't worry if you didn't fully understand the example; you can find a more detailed explanation about opening/closing windows in the Window API section.

In some cases, you may want to close the application *without sending any close signal*; to do so, you can use `App.quit()` as follows:


```
var gui = require('nw.gui');
gui.App.quit();
```

Registering system-wide hotkeys

`App.registerGlobalHotKey()` gives you the ability to register a combination of keys (**system-wide hotkeys**), which triggers an action on the application *even when it doesn't have focus, is minimized, or resides in the system tray*. This functionality leverages the power of the **Shortcut API**, which we are going to describe here for convenience. Here's a simple example:

```
var gui = require('nw.gui');
var shortcut = new gui.Shortcut({
  key : "Ctrl+Alt+A",
  active : function() {
    gui.Window.get().show();
    console.log("You pressed: " + this.key);
  }
});
gui.App.registerGlobalHotKey(shortcut);
```

As illustrated, we created a keyboard shortcut and then registered it with `App.registerGlobalHotKey()`. In the shortcut options, we've set an `active` callback, which is fired when the right combination of keys is typed down. In our example, when you press `Ctrl + Alt + A`, the main window is shown (even if not focused or is minimized) and the key combination is logged in to the console.

 **Be really careful** when registering system-wide hotkeys as you might intercept key combinations that are intended to fulfill other purposes (for example, `Ctrl + C`).

You might also register the keyboard shortcut without specifying an action and listen for the active event as follows:

```
shortcut.on('active', function() {  
    // Do something  
});
```

Otherwise, unregister it with `App.unregisterGlobalHotKey(shortcut)`.

Other app APIs

NW.js provides a few other APIs related to the application. Let's check them out:

- `App.fullArgv`: This lets us get the full list of arguments passed to the executable, including those specific to NW.js.
- `App.createShortcut(string filePath)`: This is available only on Microsoft Windows, enabling us to create a link to the application with the *AppUserModelID* set in the manifest file.
- `App.clearCache()`: Clear the HTTP cache in memory and the one on disk. This method call is synchronized.
- `App.crashBrowser()`, `App.crashRenderer()`: `App.crashBrowser()` crashes the browser and `App.crashRenderer()` crashes the renderer process to test the **Crash dump** feature (*Chapter 8, Let's Debug Your Application*).
- `App.setCrashDumpDir(string dir)`: This sets the directory where the Crash dump file will be saved (*Chapter 8, Let's Debug Your Application*).
- `App.getProxyForURL(string url)`: This lets you query the **proxy** to be used for loading the URL in the DOM.
- `App.setProxyConfig(string config)`: This sets the proxy to be used for loading the URL in the DOM.
- `App.addOriginAccessWhitelistEntry(sourceOrigin, destinationProtocol, destinationHost, allowDestinationSubdomains)`: This lets you add an entry to the whitelist used for controlling **cross-origin access**.
- `App.removeOriginAccessWhitelistEntry(sourceOrigin, destinationProtocol, destinationHost, allowDestinationSubdomains)`: This removes an entry from the whitelist used to control cross-origin access.
- `App.on('reopen', callback)`: This works only on Mac OS X; it's fired when the user clicks the dock icon for an already running application.

The Window API – working with windows on NW.js

On NW.js, the **Window API** is nothing more than a wrapper of the DOM's window object giving an additional layer of functionality to play with the application windows. Many of the window methods described are inherited by the DOM window object. As we stated earlier, window is also an instance of *Node.js EventEmitter* giving you the ability to listen and respond to window events such as `move` or `resize`.

Instantiating a new window object

There are many ways to instantiate a window object. When you need to refer to the current window, all you have to do is to call the `Window.get([window_object])` method leaving the attribute empty, as follows:

```
var gui = require('nw.gui');
var currentWindow = gui.Window.get();
```

In order to instantiate a different window object, you can proceed with passing the DOM window object to it through the `get` method:

```
var gui = require('nw.gui');
var extraWindow = gui.Window.get(
    window.open('window.html');
);
```

Otherwise, use the `Window.open(url[, options])` method:

```
var gui = require('nw.gui');
var extraWindow = gui.Window.open('window.html', {
    position: 'center',
    width: 640,
    height: 480,
    focus: true
});
```

As illustrated, the `open` method has an optional attribute, which allows you to set all the options of the window. You can find a reference of all the available settings in *Chapter 6, Packaging Your Application for Distribution*, as they are the same used in the manifest file.

There are also some custom options that can be used with the open method:

- `new-instance: true`: This will open the window in a new WebKit process.
- `inject-js-start: path/to/script.js` or `inject-js-end: path/to/script.js`: This will run the provided JavaScript code before any other DOM is constructed or after the document object is loaded but before the `onload` event is fired.



One thing to remember is that when opening a new window, it might not be focused by default; in order to give it focus, you'll have to set the focus options to `true`.



When a NW.js application starts, it may take a bit between the time when the main window is displayed and the time when the code is evaluated. In order to give a better user experience, *you'd rather hide the main window until all the background jobs have been completed*. To achieve this, you'll have to hide the application editing the manifest file as follows:

```
{
  "window": {
    "show": false
  }
}
```

Then, you can exploit the `onload` event:

```
var gui = require('nw.gui');
window.onload = function() {
  gui.Window.get().show();
}
```

That will eventually show the window when everything is fully loaded. Another use for this technique is for creating background applications that live in the system tray.

There's one last thing. As stated in the introduction, window is nothing more than a wrapper of the DOM window object. In order to access the original window, you can leverage the `Window.window` property. This is pretty convenient because the window object offers many functions that are not accessible through the Window API. We can use the case of `window.navigator.language` (the locale of the current system) in order to access it from the window object; you would proceed as follows:

```
var gui = require('nw.gui');
var currentWindow = gui.Window.get();
var language = currentWindow.window.navigator.language;
console.log(language);
```

In my case, the **it-IT** string will be logged. In the preceding example, calling `currentWindow.window` is actually the same as calling `window` directly in the current scope, but I'm sure you got the point.

Window – setting size and position of windows

There are many properties and method to size and move the NW.js application's windows. The easiest method to remember is that of accessing the `x`, `y`, `width`, and `height` properties of the window object, which is as follows:


```
var gui = require('nw.gui'),
    currentWindow = gui.Window.get();

// Log the current size and position
console.log({
  'x': currentWindow.x,
  'y': currentWindow.y,
  'height': currentWindow.height,
  'width': currentWindow.width
});

// After two seconds alter the window size and position
setTimeout(function () {
  currentWindow.x = 10;
  currentWindow.y = 10;
  currentWindow.width = 200;
  currentWindow.height = 200;
}, 2000);
```

The `Window.x` and `Window.y` properties refer to the absolute position of the window on the screen, while `Window.width` and `Window.height` are self-explanatory.

You can also prevent the window from being resized (at least by the users) by setting the `Window.setResizable(false)` method or by setting the maximum and minimum size boundaries with `Window.setMaximumSize(maxWidth, maxHeight)` and `Window.setMinimumSize(minWidth, minHeight)`.

 Do not use `setMaximumSize()`/`setMinimumSize()` together with `setResizable(false)`, or else none of them will work.

The following are a few other methods you can use in order to move or resize your application windows:

- `Window.setPosition(String position)`: This allows you to set a relative position for the window. At the moment, the only one that will work on all the platforms is `center`.
- `Window.moveTo(x, y)`: This sets the window coordinates onscreen.
- `Window.moveBy(x, y)`: This lets you move the window by the given amounts in pixels.
- `Window.resizeTo(width, height)`: This lets you set the window size.
- `Window.resizeBy(width, height)`: This lets you resize the window by the given amounts.

There are also two events that are triggered when you change the position or size of your window:

```
var gui = require('nw.gui'),
    currentWindow = gui.Window.get();

// Log new position on change
currentWindow.on('move', function (x, y) {
  console.log('New position', 'x:' + x + ', y:' + y);
});

// Log new size on change
currentWindow.on('resize', function (x, y) {
  console.log('New size', 'x:' + x + ', y:' + y);
});
```

When changing one of the values, or resizing or moving the window, either manually or through the APIs, one of the two events will be fired.

Changing the window status

Every desktop window may have a different status: *minimized*, *maximized*, *hidden*, *focused*, *blur*, or *closed*. Through the Window API, you can easily set the window status at runtime. Let's see how:

```
<script>
var gui = require('nw.gui');
var currentWindow = gui.Window.get();
var newWindow = gui.Window.open('options.html');
</script>

<a href="#" onclick="newWindow.minimize()">Minimize Window</a><br>
<a href="#" onclick="newWindow.restore()">Restore Window</a><br>

<a href="#" onclick="newWindow.show()">Show Window</a><br>
<a href="#" onclick="newWindow.hide()">Hide Window</a><br>

<a href="#" onclick="newWindow.maximize()">Maximize Window</a><br>
<a href="#" onclick="newWindow.unmaximize()">Unmaximize Window</a>

<a href="#" onclick="newWindow.focus()">Focus Window</a><br>
```

The preceding example is a good way to deeply understand how all the windows' visibility APIs work, but bear in **mind** that there are some differences in how these APIs operate on different platforms (for example on Mac OS X, `Window.show()` will restore a minified window, while on Microsoft Windows 8.1, a black window would come out). So, it's very important that you use them in couples: *minimize/restore*, *show/hide*, *maximize/unmaximize* and do a lot of testing!

You can monitor changes on the window visibility by listening for the following events:

- `Window.on('minimize', callback)`: This is emitted when the window is minimized.
- `Window.on('restore', callback)`: This is emitted when the window is restored from the minimized state.
- `Window.on('maximize', callback)`: This is emitted when the window is maximized.
- `Window.on('unmaximize', callback)` Emitted: This is when the window is unmaximized. On some platforms, this would not be thrown if the window gets manually resized.

- `Window.on('focus', callback)`: This is emitted when the window gets focus.
- `Window.on('blur', callback)`: This is emitted when the window loses focus.

Fullscreen windows and the Kiosk mode

In order to improve the user experience on small screens, more and more applications have started to implement the ability to work **fullscreen**, one for all writing applications. On Mac OS X, for example, the native maximize button in the title bar has been completely replaced with a fullscreen button. The code to implement this functionality is very simple. You can set or get the `Window.isFullscreen` property as follows:

```
var gui = require('nw.gui'),
    currentWindow = gui.Window.get();
currentWindow.isFullscreen = true;
```

Or else, you can use `Window.enterFullscreen()`, `Window.leaveFullscreen()` and `Window.toggleFullscreen()`, which are pretty much self-explanatory. You can also listen for the `enter-fullscreen` and `leave-fullscreen` events:

```
<script>
var gui = require('nw.gui');
var currentWindow = gui.Window.get();

currentWindow.on('enter-fullscreen', function () {
    console.log('Windows has entered fullscreen mode');
});

currentWindow.on('leave-fullscreen', function () {
    console.log('Windows has left fullscreen mode');
});
</script>

<a href="#" onclick="currentWindow.toggleFullscreen()">Toggle
Fullscreen</a>
```


Another cool function is the ability to put an application in the **Kiosk mode**.

The Kiosk software is the system and user interface software designed for an interactive kiosk or internet kiosk. The Kiosk software locks down the application in order to protect the kiosk from users.

– Wikipedia

When put in the Kiosk mode, the application becomes fullscreen, and there's no way to quit using the pointing device. On Linux and Microsoft Windows, you'll be able to quit using the *Alt + F4* or *Ctrl + Alt + Delete* key combinations since, otherwise, it would be seen as a virus, while on Mac OS X, which has decent Kiosk support, you'll be almost completely locked up. You'll still have to remember to disable the screen hotspots and disable, or overwrite, system-wide hotkeys, such as spotlight, mission control, and so on.



On Mac OS X, you'll have to disable the toolbar from the manifest file in order to make the Kiosk mode work; otherwise, a fatal error will be thrown.

The implementation, as we have seen for the fullscreen API, is really simple. You can set or get the `Window.isKioskMode` property or use the `Window.enterKioskMode()`, `Window.leaveKioskMode()`, and `Window.toggleKioskMode()` methods. The fullscreen events will be fired when entering/leaving the Kiosk mode. Here's how this is accomplished:

```
<script>
var gui = require('nw.gui'),
    currentWindow = gui.Window.get();

// enter-fullscreen will be fired by Window.enterKioskMode()
currentWindow.on('enter-fullscreen', function () {
    alert('Windows has entered fullscreen mode');
});
</script>

<a href="#" onclick="currentWindow.enterKioskMode();">Toggle Kiosk
Mode</a>
```



You can enable the Kiosk mode also from the manifest file (Chapter 6, *Packaging Your Application for Distribution*).

Frameless windows and drag regions

If you need more control over window appearance, you can set the window as **frameless**, completely disabling window borders and the title bar. In order to make the window frameless, you'll have to edit the application's manifest file as follows:

```
{
  "name": "My frameless App",
  "main": "index.html",
  "window": {
    "frame": false,
    "toolbar": false
  }
}
```

Once you've done that, you'll see that there's no way to move the window, and unless that's exactly what you want to achieve, you'll have to *set a draggable region* in the viewport. Let's see how we can implement it:

```
<!-- Custom title bar -->
<span class="draggable">
  Draggable
  <a href="#" onclick="window.close()" class="exit">&#x2716;</a>
</span>

<!-- Window content -->
<p>Window content</p>

<style>
html, body {
  padding:0;margin:0;
}
.draggable {
  display: block; background: #eee; padding: 10px; cursor:
  default;
  -webkit-app-region: drag;
  -webkit-user-select: none;
}
.exit {
  float: right; color: #666; text-decoration: none; cursor:
  default;
  -webkit-app-region: no-drag;
}
.exit:hover { color: #333; }
</style>
```



Remember not to abuse `-webkit-app-region: no-drag` areas as, on some platforms, the OS will treat them as non-client frame-generating issues with `contextmenu` events.




As illustrated in the highlighted lines of code, you can access this API through the `-webkit-app-region` CSS property. In our example, we've set it to drag on the custom title bar and to no-drag on the exit button, which is inside the draggable area. This is pretty neat, right? You have certainly noticed that we've used another WebKit CSS property. I'm talking about `-webkit-user-select`, which will disallow text selection. This is a very handy property as, in native desktop applications, most of the application text cannot be selected.

The taskbar icon – get the user's attention!


Until now, we have talked about windows, but there is another part of our application that is very important, especially when the application is minified or not focused. I'm talking about the **Taskbar icon** (or the Dock icon on Mac OS), which is not to be confused with the *system tray* icon we're going to deal with later.

There are a few APIs related to the taskbar icon:

- `Window.setShowInTaskbar(bool show)`: This allows the setting of the `show` attribute to false, which will hide the icon.
- `Window.setBadgeLabel(string label)`: This is a pretty cool and mostly unknown function of NW.js. It allows the showing of a string of text inside a little badge near the application icon. It works fine in Mac OS X and Microsoft Windows, but does not work on most Linux distributions, except for Ubuntu, which can display only numeric values.
- `Window.requestAttention(number|bool count)`: Sometimes, you just need to get the user's attention when your application is minified or not focused. `Window.requestAttention()` behaves differently depending on the operating system. On Mac OS X, the application must not be focused in order to fire *requestAttention*; if the `count` parameter is set to `-1`, the icon will blink once, while if set to `1`, it will blink until you give it some attention. On Microsoft Windows, both the icon and the window will blink the number of times set in the `count` parameter, for example, `currentWindow.requestAttention(5)`. On Linux (tested on Ubuntu + Cinnamon DE), all values of `count` will work except for `0`; the icon will not blink if the window is already active.

 One more way to get user attention is to fire a **notification**. Make sure you read the relative section in *Chapter 4, Data Persistence Solutions and Other Browser Web APIs*.

- `Window.setProgressBar(number progress)`: When running time-consuming processing, it might come in handy to show a simple *progress bar* over the application icon in order to keep the user updated while they are doing something else. The `progress` attribute takes values in the range between 0 and 1, so, for example, 0.5 will be half way.

 In order for `Window.setBadgeLabel()` and `Window.setProgressBar()` to work on **Ubuntu**, you'll have to create a `nw.desktop` file, as described in *Chapter 6, Packaging Your Application for Distribution*.

Closing windows

A special note should be added to `Window.close([force])` as this method and the relative event, which we saw previously in the `App.closeAllWindows()` example, can be really handy to *save data when closing a window*. Moreover, all the shutdown work might slowdown the closing process, so, in the while, we'd rather hide the window in order to improve user experience. Here's a simple example:

```
var gui = require('nw.gui'),
    currentWindow = gui.Window.get();

currentWindow.on('close', function (event) {

    // Hide the window
    this.hide();

    // Save data
    // ...

    // Close the window
    this.close(true);

});

currentWindow.close();
```

As you may have guessed when listening for the `close` event, the `close()` method doesn't quit the window straightaway, but it will wait for the `close` method to be called with the `force` parameter set to `true`. This way, you can easily hide the window (in order to keep the user from having to unnecessarily wait), save the data, and eventually close the window for good.



Remember that the `close` event is not triggered when calling `App.quit()`. Moreover, in the preceding example, you could have checked for the event parameter value to figure out whether the exit call was coming from an actual quit request from the menu, dock, or shortcut (in that case `event == 'quit'`) or from a close window request (`event == undefined`).

The closing process provides another event, which can be used when more than one window are at play. I'm talking about the `closed` event, which is triggered when a window is closed and all the associated JavaScript objects are released:

```
var gui = require('nw.gui'),
    newWindow = gui.Window.open('window.html');

newWindow.on('closed', function () {
    newWindow = null;
});
```

In the preceding example, you can see *we have respected the good practice* of assigning `null` to an object when the object is being removed. Later in the code, we might check for the `extraWindow` variable value to be different from `null` in order to verify that it's still available.

Other Window APIs

There are a few other Window-related APIs:

- `Window.title`: This sets or gets the window title at runtime
- `Window.cookies.*`: This lets you set or get window cookies
- `Window.menu`: This associates a menu to the window (we're going to deal with it later in this chapter)
- `Window.reload()`: This reloads the window
- `Window.reloadIgnoringCache()`: This reloads the window, thus cleaning the cache
- `Window.setAlwaysOnTop()`: This sets the window at the top of all the other applications' windows

- `Window.isTransparent` and `Window.setTransparent(transparent)`: These APIs allows you to set the background of the window as transparent (for example, *Adobe Photoshop splash screen*)
- `Window.showDevTools([id | iframe, headless]), Window.closeDevTools()` and `Window.isDevToolsOpen()`: These let you open, close, or check for the visibility of **DevTools** at runtime
- `Window.capturePage(callback [, image_format | config_object])`: This takes a **screenshot** of the window
- `Window.eval(frame, script)`: This evaluates a given script in the provided frame
- `Window.zoomLevel`: This sets or gets the window zoom level (it might be useful when dealing with *4k displays*)

There are also many events we can listen for in order to implement better applications:

- `capturepagedone`: This is fired when `Window.capturePage()` succeeds; a `buffer` argument is passed
- `devtools-opened` and `devtools-closed`: This is emitted when DevTools is opened or closed
- `zoom`: This is fired when the window is zoomed; a parameter with the zoom level is passed
- `loading` and `loaded`: This is relatively emitted when the window starts to reload and when the window is fully loaded (an alternative to `window.load` that doesn't rely on the DOM)
- `document-start`: This is fired when the `document` object is available, but before any other DOM object is constructed or any script is run, a `frame` attribute will be passed if we are dealing with an `iframe`
- `document-end`: This is fired when the `document` object is fully loaded; before the `onload` event is emitted, a `frame` attribute will be passed if we are dealing with an `iframe`

The Screen API – screen geometry functions

The Screen API has been lately added in order to better *handle window sizing and positioning on single or multiple screens*. Run the following code:

```
var gui = require('nw.gui');
gui.Screen.Init(); // Screen API is a Singleton
var screens = gui.Screen.screens;
```

You'll get an array of screen objects. Each screen object has the following structure:

```
screen {
  // unique id for a screen
  id : 69673536,

  // physical screen resolution
  bounds : {
    x : 0,
    y : 0,
    width : 1280,
    height : 800
  },

  // useable area within the screen bound
  work_area : {
    x : 0,
    y : 23,
    width : 1280,
    height : 773
  },
  scaleFactor : 1,
  isBuiltIn : false
}
```

You can easily access any of these properties in order to improve the user experience. Let's say, for example, that you want to position your window in the top-right corner of the screen. On Microsoft Windows, you would probably proceed as follows:

```
var gui = require('nw.gui'),
    currentWindow = gui.Window.get();
currentWindow.moveTo(screen.width - currentWindow.width, 0);
```

But then, on Mac OS X or Linux, the window title bar would be just under the OS menu bar. We should have probably been checking first for the screen's work area. Let's try again:

```
var gui = require('nw.gui'),
    currentWindow = gui.Window.get();
gui.Screen.Init();
var screens = gui.Screen.screens;

// Let's assume we have only one display
var myScreen = screens[0];
var rightCorner = myScreen.work_area.width - currentWindow.width;
currentWindow.moveTo(rightCorner, myScreen.work_area.y);
```

Now our window is exactly in the top-right corner, without overlapping with the taskbar, but what would happen when we are attaching our laptop to a projector with a lower resolution? Well, as you may expect, our window will get out of the screen boundaries. Let's then add a `displayBoundsChanged` listener in order to keep the window in position:

```
gui.Screen.on('displayBoundsChanged', function (newScreen) {
    rightCorner = newScreen.work_area.width - currentWindow.width;
    currentWindow.moveTo(rightCorner, newScreen.work_area.y);
});
```

The `newScreen` attribute is a new screen object, which contains all the properties seen earlier in order to make changes based on need. That's it for our example, but in a real scenario, you might want also want to change window size, zoom, or move the window to a new screen. All of that is possible combining the use of screen data and the three possible events: we've already seen `displayBoundsChanged`, which is triggered when the screen resolution changes, but we also have `displayAdded` and `displayRemoved`, both passing a screen attribute.

The Menu API – handling window and context menus

In NW.js, menus can be used in three different contexts:

- **Contextual menus:** This is displayed when right-clicking an element inside the application.
- **Window menus:** On Microsoft Windows and Linux, you can have one per window; however, in Mac OS X, you can have one, which will be shown on the System taskbar, per application.

- **Tray icon menus:** This is displayed when clicking on a tray icon usually on the right side of the OS taskbar.

In this chapter, we're going to deal with the first two contexts. For tray icon menus, the same basic rules apply, but refer to the **Tray API** section to learn more about it.

The contextual menu

In order to instance a new menu on NW.js, we should proceed as follows:

```
var gui = require('nw.gui');  
var menu = new gui.Menu();
```

Once the menu has been created, we have to append one or more `MenuItem` objects to it:

```
menu.append(new gui.MenuItem({  
  label: 'Menu Item'  
}));
```

We have three different types of menu items: normal (default value), checkbox (on/off switch), and separator. Here's the full list of options that can be declared when instantiating a new menu item:

```
var menuItem = new gui.MenuItem({  
  type: 'checkbox', // Can be normal, checkbox or separator  
  label: 'Menu Checkbox',  
  icon: 'icon16.png',  
  tooltip: 'Hello World!',  
  click: function () {  
    // Do something on click  
  },  
  enabled: true,  
  checked: false, // Only for checkboxes  
  key: 'M',  
  modifiers: 'ctrl-shift',  
  iconIsTemplate: true, // Only on Mac OS  
  submenu: new gui.Menu() // Accept a Menu Object  
});
```

Most of the options are self-explanatory and can be set both as options, when instantiating the object, and as properties at runtime. The `click` option accepts a callback, which will be called when you click on the menu item and also at runtime using `MenuItem.on('click', callback)`. The modifiers (*Cmd*, *Shift*, *Ctrl*, *Alt*) and `key` (a single key) options enable you to associate a keyboard shortcut with the current menu item.



On contextual and tray menus, keyboard shortcuts associated with menu items are fired only if the corresponding menu is open. Keyboard shortcuts registered on the window menu will instead be fired even when the menu is closed. **It's very important** to remember this and even more when registering custom keyboard shortcuts using `document.addEventListener('keyup', callback)` since both the events would be fired.

Once we have seen all the menu item options, we have to show the contextual menu on some user action. In order to show the menu, you can use the `Menu.popup(x, y)` method, but as you probably don't want to manually provide coordinates but instead associate it with the *right-click* on a given element, let's see how to trigger it when the `contextmenu` event is fired:

```
<div id="area" style="padding:40px;background: #00f;">Right click
here</div>
<script>
var gui = require('nw.gui');
var menu = new gui.Menu();
menu.append(new gui.MenuItem({'label':'menuItem1'}));
document.getElementById('area').addEventListener('contextmenu',
  function(e) {
    e.preventDefault();
    menu.popup(e.x, e.y);
    return false;
  });
</script>
```

As illustrated, the pop-up coordinates are passed as arguments by the event itself and then caught by the `popup` method.

The Menu object has a few other properties and methods, which allow you to play with menu items:

- `Menu.items`: This is an array of `menuItem`s associated with the menu, which implies that you can access `menuItem` using its position in the array, as follows:
`Menu.items[i]`
- `Menu.insert(MenuItem item, number i)`: As opposed to `Menu.append()`, which appends menu items at the end of the queue, the `insert` method allows you to insert them in any given position by passing the `i` attribute corresponding to the array key (if a menu item is already present at that position, its key and the followings ones will be incremented by one).
- `Menu.remove(MenuItem item)` and `Menu.removeAt(number i)`: These allow you to remove a given menu item.

The window menu

Everything we learned about contextual menus also applies to window menus but with a few important differences. First, in order to implement a window menu, you have to set the menu type as `menubar`:

```
var gui = require('nw.gui');  
var windowMenu = new gui.Menu({ type: 'menubar' });
```

If you plan to deploy the application on Mac OS X, you'll probably want to add the following code in order to implement the **built-in Mac menu** (*app*, *edit*, and *window* menus):

```
windowMenu.createMacBuiltin("App Name");
```

The `App Name` string is provided in order to give value to a few of the submenu items. Getting back to our example, all the first-level menu items of `windowMenu` *must have a submenu* as on most platforms, empty menus are not allowed:

```
// Add first level menu  
var firstMenuItem = new gui.MenuItem({  
  label: 'File',  
  submenu: new gui.Menu()  
});  
windowMenu.append(firstMenuItem);  
  
// Add submenu items  
firstMenuItem.submenu.append(new gui.MenuItem({label: 'Open'}));
```

```
firstMenuItem.submenu.append(new gui.MenuItem({label: 'Save'}));
firstMenuItem.submenu.append(new gui.MenuItem({type: 'separator'}));
firstMenuItem.submenu.append(new gui.MenuItem({label: 'Exit'}));
```

It might look tricky, but it's actually pretty simple. We add a first-level menu called File to windowMenu and then add all the relative submenus (Open, Save, and Exit). Eventually, we can set windowMenu as the menu for the current window by just adding the following line of code:

```
gui.Window.get().menu = windowMenu;
```

That's it. Just make sure that you read the following note about Mac OS X as it might be quite tricky to get it.



When deploying your application on Mac OS X, you'll notice that the first menu always has the name of the application as set in the .plist file (we'll see that later in *Chapter 6, Packaging Your Application for Distribution*) despite the fact that you are providing a different menu name. Put your heart at peace, you can do nothing about it. Moreover, as anticipated, only one menu per application is allowed on Mac OS X applications, so, if you need to implement different menus based on the platform, you can take advantage of the `process.platform` property.

File dialogs – opening and saving files

Before talking about **file dialogs**, I have to do a little introduction. It is probably obvious, but while in the browser, file dialogs allow you to upload or download files, in NW.js, they do nothing but *pass the path of the selected folder or files*. So, any following operation will be done on the original file, not on a copy.

WebKit enables you to open file dialogs through the **file input field** but with many limitations due to *security concerns*. In NW.js, those limitations have been removed, and the default file input fields have been enhanced in order to give a full native user experience.

A faster way to open a file dialog is to add an event listener to the file input field as follows:

```
<input id="fileDialog" type="file">
<script>
document.querySelector('#fileDialog')
  .addEventListener("change", function() {
    var filePath = this.value;
```

```
        alert(filePath);
    });
</script>
```

In the preceding example, a default file input field will be rendered in the viewport, clicking on it will open the file dialog, and then selecting a file will trigger the change event, which will return the file path inside the `this.value` property. This is pretty easy, but you'll probably want to get rid of the ugly default input field. Let's see how, using a reusable JavaScript object:

```
<input id="fileDialog" type="file">
<script>
function FileDialog(inputId, callback) {
    var fd = this;
    fd.chooser = document.querySelector(inputId);
    fd.chooser.addEventListener("change", function() {
        var path = this.value;
        callback(path);
        fd.chooser.value = '';
    });
    fd.open = function () {
        fd.chooser.click();
    };
    return fd;
}

var myFileDialog = new FileDialog('#fileDialog', function (path) {
    // Do something with path
    console.log(path);
});
</script>

<a href="#" onclick="myFileDialog.open();" >Open file</a>
```

What I've done here is instance a very simple object that listens for value changes in file input fields and triggers a given callback. The input value is then reset in order to trigger again if the same file get selected twice. This is obviously one way to deal with file dialogs; you can easily write a better handler or rely on **jQuery**, as described on the wiki page at <https://github.com/nwjs/nw.js/wiki/Native-UI-API-Manual>.

Until now, we have been dealing with a single file open dialog, but we can take it a step further by opening *multiple files*, *directories*, or a **save dialog**. You can find a reference to the many available options in the following sections.

Opening multiple files

Here's an example of opening multiple files:

```
<input type="file" multiple />
```

Using the code provided, the path attribute will return a list of paths separated by a "," character.

Filtering by file type

Here's an example of filtering by file type:

```
<input type="file" accept=".doc,.docx,.xml,application/msword">
```

This way, the file dialogs will allow you to restrict the kind of files that can be opened by your dialog (it doesn't work with save dialogs).

Opening a directory

Here's an example of opening a directory:

```
<input type="file" nwdirectory />
```

All the files will be disabled, and you'll be able to select only directories.

Saving files

Here's an example of saving files:

```
<input type="file" nwsaveas />
```

Instead of the default open dialog, a save dialog will be shown with the ability to type a filename. You can also suggest a default filename as follows:

```
<input type="file" nwsaveas="defaultFileName.txt" />
```

Suggesting a default path

Here's an example of suggesting a default path:

```
<input type="file" nwworkingdir="C:\Documents" />
```

Your application may wisely suggest a default path to open or save files. The path must be clearly provided accordingly to the operating system.

Opening files through file dragging

Many native desktop applications provide an alternative way to open files by just dragging them over the application. In order to implement this functionality, we can take advantage of **HTML5 APIs**. As this is actually beyond the purpose of the book but still very useful, I will report an example to simplify the task:

```
<div id="dropFileHere">Drop File Here</div>

<style>
#dropFileHere { display: block;background: #f00; }
#dropFileHere.hover { background: #0f0; }
</style>

<script>
  // Prevent default drop file behaviors
  window.ondragover = function(e) { e.preventDefault(); };
  window.ondrop = function(e) { e.preventDefault(); };

  var holder = document.getElementById('dropFileHere');
  holder.ondragenter = function() { this.className = 'hover'; };
  holder.ondragleave = function() { this.className = ''; };
  holder.ondrop = function(e) {
    e.preventDefault();
    this.className = '';
    for (var i = 0; i < e.dataTransfer.files.length; ++i) {
      var filePath = e.dataTransfer.files[i].path;
      // Do something with filePath
    }
  };
</script>
```


The Tray API – hide your application in plain sight

The **system tray** is a small area of the screen, usually placed on the right-hand side of the system taskbar, which contains a set of icons belonging to long-running applications or services that don't fit into the *taskbar*, which, otherwise, will be overcrowded.

In order to associate one or more tray icons to your application, you'll have to instance a **Tray object** as follows:

```
var gui = require('nw.gui');
var tray = new gui.Tray({
  icon: 'icon.png'
});
```

On **Microsoft Windows** and **Linux**, once you have an instance of the tray, you can add a tooltip assigning the `Tray.tooltip` property, change the icon at runtime using `Tray.icon`, or assign a menu to `Tray.menu`.

 The NW.js Tray API doesn't work on all Linux distributions as, at the moment, it is still using `GTKStatusIcon` instead of the newer `AppIndicator`. However, there's an open issue on GitHub, so this might change in future releases.

In order to show the menu, you'll have to right-click on the icon. When clicking instead with the left button, a `click` event will be generated. Let's see an example for clarity:

```
var gui = require('nw.gui');
var tray = new gui.Tray({
  icon: 'icon16.png'
});

// Assign tooltip
tray.tooltip = 'Hello World!';

// On right click show menu
tray.menu = new gui.Menu();
tray.menu.append(new gui.MenuItem({ label: 'Exit' }));

// On left click change icon
// You'd get the same behavior using alticon when a menu is attached
// to the tray icon
tray.on('click', function () {
  tray.icon = 'icon16_clicked.png'
});
```


You can also remove the tray icon using the `Tray.remove()` method, but remember to assign a null value to the object after running it.

```
tray.remove();  
tray = null;
```



When you instance a new tray object, remember to do it inside the global scope, or else, another scope, which will exist during all the life of the application; otherwise, as the garbage collection runs, the icon will disappear.

On **Mac OS X**, the procedure to set a tray icon is pretty much the same, but you can leverage a few more properties and methods. Moreover, when you assign a menu to the tray icon, the `click` event *won't be fired* as the left-click will be used to show the menu itself.

As it's beyond the purpose of the book to dive deep into Mac OS X custom features, I will list them here for reference:

- `Tray.title`: This lets you set or get custom text to show beside or instead of the icon.
- `Tray.alticon`: This lets you get or set an active icon shown when the menu attached to the tray icon gets opened.
- `Tray.iconsAreTemplates`: On Mac OS X, icons can be treated as templates (true by default).



In order to load retina icons when needed, you can proceed as follows:

```
if (window.devicePixelRatio > 1){  
    tray.icon = 'icon16@2x.png'; // The icon size should  
    be 32x32px  
}
```

Otherwise, you can combine multiple PNG images into a tiff icon using the following terminal script (only working on Mac OS X):

```
$ tiffutil -cathidpicheck icon16.png icon16@2x.png -out  
icon16.tiff
```

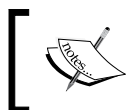
There's one last thing. It's not possible to customize the look of a menu attached to a tray icon; however, you could show a custom-shaped transparent window tracking the position of the click. In order to do so, when listening for the click event, you can check for the `position` object passed as an argument. Check the following example for reference:

```
tray.on( 'click', function(position) {
  // position.x is x coordinate of the tray icon
  // position.y is y coordinate of the tray icon
});
```

The Clipboard API – accessing the system clipboard

NW.js' **Clipboard API** lets you read and write plain text from the *system clipboard*. There are three main methods to access the content of the clipboard. `Clipboard.set(data)` enables you to copy some text data into the clipboard, the `Clipboard.get()` method allows you to retrieve that data, and `Clipboard.clear()` allows you to clear it up. The following is a simple example:

```
var gui = require('nw.gui'),
    clipboard = gui.Clipboard.get();
// Copy
clipboard.set('I love NW.js :)' );
// Paste
var data = clipboard.get('text');
// Clear
clipboard.clear();
```



The NW.js Clipboard API is still pretty young. At the moment, you can only access plain text and only from the system clipboard. The selection clipboard in X11 is not supported.



In order to do cut, copy, and paste, you can rely on `document.execCommand('cut')`, `document.execCommand('copy')` and `document.execCommand('paste')`.

The Shell API – platform-dependent desktop functions

The **Shell API** is made of a set of methods that allow you to leverage platform-related tasks:

- `Shell.openExternal(String URI)`: This will open any given URI (`http://`, `mailto:`, and so on) with the default system application if one has been associated; otherwise, nothing will happen.
- `Shell.openItem(String file_path)`: This will open any given file with the default system application if one has been associated; otherwise, the **Open with** dialog will be shown.
- `Shell.showItemInFolder(String path)`: This will open the given path inside the system file explorer/finder.

Here's a simple example to open `Google.com` on your default browser:

```
var gui = require('nw.gui');  
gui.Shell.openExternal('http://www.google.com');
```

Summary

In this chapter, I did my best to introduce you to the use of **NW.js Native UI APIs**. We played with windows, menus, tray icons, and many more OS-related functions. I often used a cookbook style because I firmly believe that *the best way to learn is by example*, the way we learned when we were little kids. Unfortunately, I could not dwell on all the APIs, but I tried to deal with those that I consider most important for the purposes of the book.

Now that you have a decent understanding of how to interact with your application GUI, we can move to the next topic – **Node.js**. You're probably already experts in the field but, in the context of NW.js programming, Node.js must be approached in a slightly different way than usual. You will learn how to target and deal with *path processing* on multiple platforms, use *modules*, and *pass objects between WebKit and Node.js*. Let's not waste any more time and move on to *Chapter 3, Leveraging the Power of Node.js*.

3

Leveraging the Power of Node.js

Node.js is an **event-driven runtime environment** created by Ryan Dahl in 2009 that provides a proper environment to handle tens of thousands of concurrent connections without worrying about their operational costs.

As it also provides a *built-in asynchronous I/O library for file, socket, and HTTP communication*, it has been adopted as a server-side platform by many of the top players of the industry, such as Yahoo, Microsoft, Groupon, and LinkedIn.

Its daily usage in simpler scenarios is probably justified by the fact that Node.js applications are written in **plain JavaScript**, a well-established web programming language with a lot of documentation and a steep learning curve.

Many of you have been using Node.js to build **client-server** applications and might be tempted to reproduce client/server architecture in NW.js. That's not totally wrong; you can port your Node.js web application directly into NW.js with very little effort, but *it would affect the platform potential*.

You should always remember that NW.js lets you work with Node.js modules *directly from the DOM*. It will make your applications faster and shorten your code from tons of useless lines of code.

In this chapter, I will try to explain the peculiarities of Node.js programming within the NW.js system, dwelling on *application architecture, path handling, context issues, and modules*. I will not dwell on actual Node.js programming since you're probably more experienced than me on the subject.

Routing and templating in NW.js

Many Node.js developers will be tempted to adopt web frameworks such as *Express* in order to achieve routing and templating for their applications but, as we stated in the introduction, that would be not recommended as it's the client-server way to approach the problem. Let's see how to achieve the same result **the NW.js way**.

If you think about it, most of the recent **web application frameworks** provide client-based routing. When you click on a link, the page won't reload; instead, the new view will be rendered in real time and the URL of the page will change accordingly thanks to the HTML5 History API. This way, a minimal amount of data will be exchanged between the server and the client, and you won't feel the bad experience of the full page reloading.

So, you can obviously rely on a web application framework, such as **AngularJS** or **Ember.js**, to do all of your routing, but you can also write a very simple view loader yourself as, in a desktop application, you won't need to change URLs:

```
<div id="mainView"></div>
<script>
var myApp = {};
myApp.loadView = function (view, containerId, callbackName) {
  var fileName = 'views/' + view + '.html';
  $(containerId).load( fileName , function (response, status) {
    if (status === 'error'){
      console.warn('Could not load ' + fileName);
    } else if (typeof myApp[callbackName] === 'function') {
      myApp[callbackName] ($(this));
    }
  });
};
// Loading views/view1.html
myApp.loadView('view1', '#mainView', 'view1Controller');
myApp.view1Controller = function ($currentView) {
  // Do something after main view is loaded
};
</script>
```



The reported examples are purely for educational purposes in order to understand how you might proceed in the absence of a web application framework. In this chapter, I'm going to use **jQuery** to shorten code a bit, but you can obviously go vanilla!

In the preceding example, we loaded the `views/view1.html` file at runtime, providing a callback function in order to run a piece of code only once the view has been loaded. This example highlights another major drawback of loading full pages; you will lose the window context at each call.

There's not much to add about **templating**. If you don't want to rely on a web application framework but rather keep using a *Node.js template engine*, you can easily do it directly from the DOM. Let's review the `loadView()` function of the previous example in order to load **Swig templates**:

```
myApp.loadSwigView = function (view, data) {
  var fileName = 'views/' + view + '.html',
      swig = require('swig');
  return swig.renderFile(fileName, data || {});
};
// Loading views/swigTemplate.html
var template = myApp.loadSwigView('swigTemplate', {
  pagename: 'awesome people',
  authors: ['Paul', 'Jim', 'Jane']
});
$('#mainView').html(template);
```

As illustrated, this is really simple; we load `views/swigTemplate.html`, render the data through the Swig template engine, and append the output to the `#mainView` div. All the work is done directly in the DOM without messing with cumbersome client/server transactions.



If you are planning to develop a large desktop application, I would personally recommend that you go with a web application framework, with all the advantages of the MVC pattern. However, different scenarios require different strategies, so the choice is up to you.

Node.js global and process objects

In NW.js, the `global` object works much like it does in Node.js with the peculiarity of being accessible from within the DOM, giving an additional way to *share objects between the window and Node.js contexts*. Moreover, when the `window` object is created, all the members of the `global` object are assigned to it, which is why you can use `require()` from within the DOM.

This binding works both ways, so you can access global properties and methods assigned in the window context inside Node.js modules. Let's see a simple example:

- **index.html:**

```
var gui = require('nw.gui');
gui.Window.open('newWindow.html');
global.sharedObjects = {
  myData: 'Foo'
};
var myModule = require('./module.js');
```

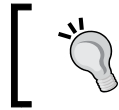
- **module.js:**

```
console.log(sharedObjects.myData);
```

- **newWindow.html:**

```
console.log(global.sharedObjects.myData);
```

As illustrated, inside our module, we've omitted `global` as it's already assigned as the **global root variable**. You may be tempted, then, to do all of your coding inside Node.js modules, but *remember that a few DOM operations cannot be run in the Node.js context*; for example, `window.openDatabase` and `nw.gui` won't work as they depend on the window context.



You should remember that overusing global variables is usually considered bad practice, so think twice—almost certainly, there's a better way to handle it.

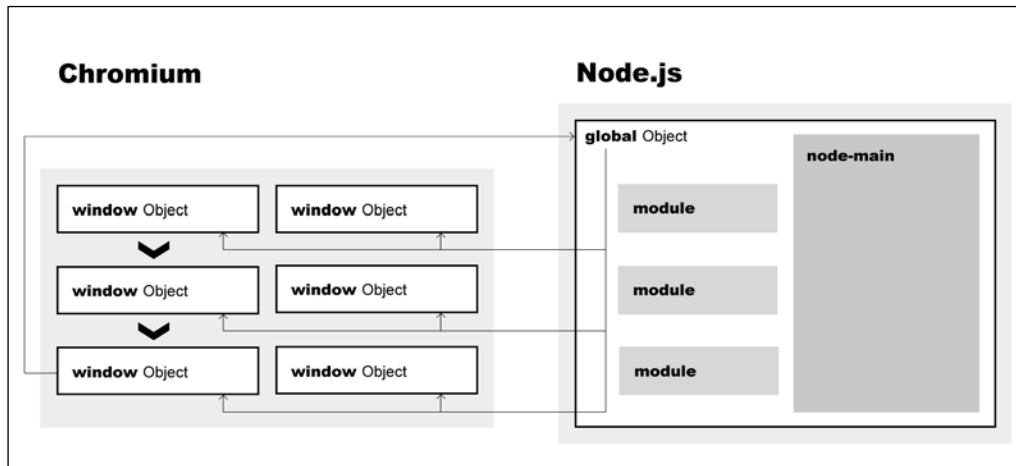
The `process` object is a global object that can be accessed from both the window and Node.js contexts. In NW.js, some new fields have been added to it:

- `process.versions['NW.js']`: This gets the NW.js version
- `process.versions['chromium']`: This gets the NW.js chromium version
- `process.mainModule`: This gets the start page (such as `index.html`) specified in the manifest file's `main` field

If `node-main` has been specified in the manifest file, `process.mainModule` points to the specified file.

The window object

A special note should be added on how the window object is handled by Node.js.



When all the initial stuff has been done, as soon as it's ready, the window object is passed to Node.js inside the global object so that you can reference it in a module as `global.window` or simply `window`. The window reference is then *replaced with a new one on page navigation*. Check the following example:

- **nodeModule.js:**

```
exports.checkWindowVar = function () {
  console.log(window.myVar);
}
```
- **index.html:**

```
var myVar = 'Hello';
var nodeModule = require('./nodeModule.js');
nodeModule.checkWindowVar(); // Hello
// Navigate to newWindow.html
location.href = 'newWindow.html';
```
- **newWindow.html:**

```
var myVar = 'Foo';
var nodeModule = require('./nodeModule.js');
nodeModule.checkWindowVar(); // Foo
```


The preceding code will always log the last instance of `myVar`. However, if, instead of navigating to the new page, you *open it in a new window*, the reference is kept with the first instance even if the window has been closed. So, we edit `index.html`:

```
var myVar = 'Hello';
var gui = require('nw.gui');
gui.Window.open('newWindow.html');
```

When you check the console, you will still get `Hello`. In this case, if you need to access the new window object from Node.js, you'll have to manually replace it. Let's edit `newWindow.html` accordingly by just adding `global.window = window`; at the beginning of the code. You will get your `Foo` string back as expected.

Using NW.js' main module

As we stated in the first chapter, NW.js checks for the `main` field in the manifest file in order to load the given file when opening the application's first window. The `node-main` field acts similarly, but instead of referring to an HTML file, it will allow you to *set a main Node.js module*, which will run in the background through the application's life cycle independently of the window context (even though it will quit by default when all the application windows are closed).

In the previous chapter, we've seen that we can use `process.mainModule` from the window context in order to get the main module, but more specifically, besides `global`, we can take advantage of `process.mainModule.exports` to share objects between the two contexts.



Thanks to the `global` object, the window object is also shared with the main module, but you can't obviously access it until a ready signal is sent from the DOM.

Let's see an example:

- **package.json:**

```
{
  "name": "secondsSinceStart",
  "node-main": "nodemain.js",
  "main": "index.html"
}
```

- **nodemain.js:**

```
var seconds = 0;
setInterval(function() {
```

```
        seconds++;
    }, 1000);
    exports.secondsSinceStart = function () {
        return seconds;
    }
}
```

- **index.html:**

```
<div id="time"></div>
<script>
var seconds = process.mainModule.exports.secondsSinceStart();
$('#time').text(seconds + ' seconds');
</script>
```

We've set the manifest file to load `nodemain.js` as the main module; when you reload the window, you'll find that the `seconds` variable hasn't been reset. That's because the main module keeps running in the background independently of the window instance.

Handling paths in NW.js

At first, handling paths in NW.js can be quite tricky. The first time you might find yourself in trouble is when requiring a custom module not stored in the default `node_modules` folder. Let's say, for example, this is our folder structure:

```
Application root
- data/myData.json
- js/script.js
- modules/myModule.js
- modules/otherModule.js
- index.html
- package.json
```

When requiring a module from the window context, the module's path is treated as relative to the application's root directory, so in our example, whether you're calling it from `index.html` or `js/script.js`, we would proceed as shown in the following code:

```
require('./modules/myModule');
```

When calling a module from another module, however, you should use the relative path from the current file. So, let's say we're requiring `otherModule.js` from `myModule.js`, which is in the same folder:

```
require('../otherModule');
```

The next thing you might worry about is how to *access files for I/O operations* from both contexts using their relative path. That's actually pretty straightforward as you can always consider the file path as relative to the application root folder:

```
var fs = require('fs');
fs.readFile('data/myData.json', 'utf8', function (err, data) {
  console.log(JSON.parse(data));
});
```

Whether you are using this code inside `index.html`, `js/script.js` or `modules/otherModule.js`, the relative file path will always be the same.

There are two ways to get the **absolute path** of the application root folder. You might, of course, call `window.location.pathname` on `index.html`, but there's a much better Node.js way to do it. Node.js provides a `__dirname` global object, which returns the current file pathname with the single *limitation to be accessible only from the Node.js context*. However, in order to use `__dirname` inside the window context, we can provide a simple workaround. We can create a `util.js` module inside the application root folder:

```
exports.dirname = __dirname;
```

We can then require the module from `index.html`:

```
var dirname = require('./util').dirname;
console.log(dirname);
```

As the `util` module is in the root folder, `dirname` will always return the application root path wherever it is called.


One last thing to consider about paths is that in Microsoft Windows, backslashes are used instead of common slashes. In order to provide the right path format for each platform, you should make use of the Node.js **path module**:

```
var path = require('path');
console.log(path.normalize('/my/custom/folder'));
```

On Mac OS X or Linux, `path.normalize()` will return `/my/custom/folder`, but on Microsoft Windows, it will return `\\my\\custom\\folder` (the extra backslashes are for escaping). If you need to join multiple paths, you can use `path.join()` instead:

```
var path = require('path'),
    dirname = require('./util').dirname;
console.log(path.join(dirname, 'data/myData.json'));
```

That, based on the platform, might return `../myApp/data/myData.json` or `c:\\...\\myApp\\data\\myData.json`.

 If you need to retrieve the application's absolute **dataPath**, you can use the `App.dataPath` API as described in *Chapter 2, NW.js Native UI APIs*.

If you need to target specific folders based on the current operating system, you can take advantage of the `process.platform` property:

```
var path = require('path'),
    desktopPath;
switch(process.platform) {
  case 'darwin':
  case 'linux':
    desktopPath = path.join(process.env.HOME, 'Desktop');
    break;
  case 'win32':
  case 'win64':
    desktopPath = path.join(process.env.USERPROFILE, 'Desktop');
    break;
  default:
    throw 'Platform not supported';
}
console.log(desktopPath);
```

This is a very raw example of how to return the current platform's *desktop path*.

NW.js context issues

We've already talked about contexts and how to pass objects between the window and Node.js contexts using `global`, but there are still a few things we need to consider when developing on multiple contexts.

As first, you should consider *that switching between the DOM and the Node.js context takes a while*, so if you're planning to run hundreds of concurrent operations that depend on both the window and Node.js contexts, you might get a slight delay. In order to avoid this kind of issue, you'd better choose in advance which context to adopt.

One more thing that might be very confusing at first is that the same objects from different contexts also have different constructors in their prototype chain. That results in misbehavior of the `instanceof` operator, which *will not be able to recognize prototypes coming from a different context*.

Let's check an example:

- **nodeContext.js:**

```
exports.myArray = ['one', 'two', 'three'];
```

- **index.html:**

```
var windowContext = {
  myArray: ['one', 'two', 'three']
};
// First statement
console.log(windowContext.myArray instanceof Array); //
  true
var nodeContext = require('./nodeContext');
// Second statement
console.log(nodeContext.myArray instanceof Array); // false
```

As illustrated, we're getting different results despite it looking like we're checking for the very same array object. A faster way to avoid this kind of problem is to get rid of `instanceof` in favor of a more reliable alternative, such as `Array.isArray`. If you can't afford it, you can take advantage of the **nwglobal** module (<https://github.com/Mithgol/nwglobal>).

A very simple way to export Node.js constructors is provided by **nwglobal** in order to avoid `instanceof` issues in both the window and Node.js contexts. Let's review the second statement of the previous example:

```
var nwglobal = require('nwglobal');
console.log(nodeContext.myArray instanceof nwglobal.Array); //true
```

This time, the statement will return `true` as we get the right constructor with `nwglobal.Array`.

Thanks to **nwglobal**, we can also create objects using Node.js constructors, so if you need to pass an object to a Node.js module that makes use of `instanceof`, you can proceed as shown in the following code:

- **nodeContext.js:**

```
exports.checkArray = function (myArray) {
  return myArray instanceof Array;
}
```

- **index.html:**

```
var windowArray = ['one', 'two', 'three'];
console.log(nodeContext.checkArray(windowArray)); // false
var nodeArray = nwglobal.Array('one', 'two', 'three');
console.log(nodeContext.checkArray(nodeArray)); // true
```

One last thing to note is that `nwglobal` enables you to do **array casting**. That's really convenient if you need to pass a jQuery set to a Node.js module.

Let's review our code:

```
var nodeArray = nwglobal.Array.prototype.slice.call(windowArray);
console.log(nodeContext.checkArray(nodeArray)); // true
```



You can find more information about context issues on NW.js at <https://github.com/nwjs/nw.js/wiki/Differences-of-JavaScript-contexts>.

If you are using **RequireJS** inside your NW.js project, it will conflict with the Node.js `require`. In order to fix it, you can take a look at <http://www.requirejs.org/docs/faq-advanced.html>.

Working with Node.js modules

One of the strengths of Node.js is surely the availability of tens of thousands of modules that are freely downloadable with **npm**. As you probably know already, Node.js supports three kinds of modules:

- Internal modules, which are parts of Node API
- Third-party modules written in JavaScript
- Third-party modules with C/C++ add-ons

You can use all of them in NW.js with some precautions.

Internal modules

Internal modules such as `fs`, `path`, or `http` are available out of the box and you can call them very easily with the following code:

```
var fs = require('fs');
```

Third-party modules written in JavaScript

Third-party JavaScript modules can be installed with npm and will be downloaded with all their dependencies inside the `node_modules` folder of the application's root folder in order to be shipped with the application. You can require them very easily using the following code:

```
var modulename = require('modulename');
```

Third-party modules with C/C++ add-ons

Even though most Node.js modules are written in plain JavaScript, you might require a module containing C/C++ add-ons. In this case, the situation is slightly different to the previous example as, when installed, these modules are also compiled so that they work with Node.js.

Since NW.js has a different **Application Binary Interface (ABI)** than Node.js, the compiled version will not be compatible with your application.

You'll have to manually compile the module using **nw-gyp**, a special fork of *node-gyp*. Let's see how to proceed on this course:

1. Install **nw-gyp** with the following command:

```
npm install nw-gyp -g
```



On Mac OS X, the only requirement is to install Xcode command-line tools; on other operating systems, you'll have to manually install Python, Make, and a proper C/C++ compiler. You can find more information about this at <https://github.com/nwjs/nw-gyp>.

2. Install your module as usual with npm.
3. Run **nw-gyp** in the module folder (where the `binding.gyp` file is located), providing the NW.js target version:

```
$ nw-gyp rebuild --target=0.11.2
```

4. That's it. Now you can require your module as usual:

```
var modulename = require('modulename');
```



On Microsoft Windows, the engine's executable file must have the name `nw.exe` for C/C++ add-ons to work as they use the NW.js engine to sustain their execution.



You can install NW.js modules either using `npm install module` or by adding modules as dependencies in your manifest file and then running `npm install`. A simple example is as follows:

```
{
  "name": "hello-world",
  "main": "index.html",
  "dependencies": {
    "markdown": "0.5.x",
    "swig": "1.4.2"
  }
}
```

Summary

As I stated in the introduction, this chapter was all about adapting your previous Node.js knowledge to NW.js development. This requires a good deal of *flexibility* and the difficult task of leaving tried and tested programming habits, but it will result in faster, better structured, and platform-optimized code.

Some of the concepts I've introduced might be very simple to the most skilled of you, but I would prefer to be tedious rather than leaving it to chance.

In the next chapter, we're going to deal with **Browser APIs**, more precisely, with all the advantages they can bring to your NW.js applications, with a focus on how to store persistent data thanks to *Web SQL Database*, *embedded databases*, and *Web Storage APIs*.

4

Data Persistence Solutions and Other Browser Web APIs

Lately, when talking about **Browser Web APIs**, we often refer to them with the general term **HTML5 APIs**. This is due to the fact that the new HTML5 standard has brought a breath of fresh air in web application development. Things like WebSockets, Canvas, video, and web storage were cumbersome and difficult to achieve before the advent of HTML5.

Actually, Browser Web APIs are a set of APIs that include *standard JavaScript APIs, XMLHttpRequest, WebKit APIs, V8 APIs, HTML5, and other emerging APIs*.

As you can imagine, Chromium Web APIs give you an incredible number of tools to build your applications, from custom input fields (date, e-mail, and so on) to real-time push notifications through WebSockets. However, for the sake of brevity, in this chapter, we're going to deal only with those APIs that come in particularly handy in NW.js desktop application development. We will walk through the various solutions to *store permanent data, deal with media files, shed some light upon security issues*, and eventually talk about the *Web Notification API*.

Maybe you've already considered it, but there's an amazing thing you should remember when developing your NW.js application. Even though you're using JavaScript, CSS, and HTML, as you would when developing a web application, here you're actually developing for only one browser, **Chromium**. That means you can forget about browser compatibility and use the latest implementation of Browser Web APIs. There will be no more prefixes (apart from WebKit) and no more polyfills, and don't be ashamed of using edge technologies such as notifications, flex, canvas, and even geolocation.

Data persistence solutions

NW.js provides many ways to handle data persistence. You can obviously rely on a Node.js persistent database, such as *EJDB*, *locallyDB*, *NeDB*, or *LowDB*, but in this chapter, we're going to see how we can save and access persistent data by leveraging some of the latest browser storage technologies.



I will not dwell on **HTML5 Application Cache** as I find its use to be pointless in a NW.js application; however, if you need to implement offline navigation, you know you can rely on it as well.

Web storage

Web storage is a browser technology, which was once part of the HTML5 specification but lately split into its own specification, used to store temporary or permanent **key-value** data on the client side. It's by far *the easiest way to achieve data persistence* in the browser but also one of the weakest. Let's see some disadvantages of web storage:

- Maximum 5 MB storage limit
- Web storage calls are synchronous and can block the main document from rendering
- As web storage writes data to the hard disk, its performance is pretty bad
- No advanced indexing/queries as *the stored values can only be of the string type*

It's pretty obvious that web storage can be your technology of choice only if you're planning to make very simple and non-intensive data transactions.

There are two types of web storage. The first one, termed **localStorage**, is permanent and lasts beyond the application session; the latter type, termed **sessionStorage**, is temporary and gets deleted when the application is closed. In this chapter, we are going to deal with `localStorage`.

In order to store data in its key-value structure, you should proceed as follows:

```
localStorage.setItem('myKey', 'myValue');
```

Then, if you need to get the value back, you can easily access the stored string by its **key**:

```
var foo = localStorage.getItem('myKey');  
console.log(foo); // myValue or null if not set before
```

The preceding examples could be rewritten using the square bracket or dot syntax:

```
localStorage.myKey = 'myValue';  
console.log(localStorage['myKey']); // myValue
```

As you can see, working with web storage is really simple; to delete a key-value, you can use the following code:

```
localStorage.removeItem('myKey');  
// or  
delete localStorage.myKey;
```

You can even delete all the stored data using `localStorage.clear()`.

The `localStorage` object also has a `length` property, which returns the number of key-value pairs stored in it, and a `key()` method, which gets a given value by its numeric key:

```
var lastKey = localStorage.length - 1;  
console.log(localStorage.key(lastKey)); // Returns older value
```



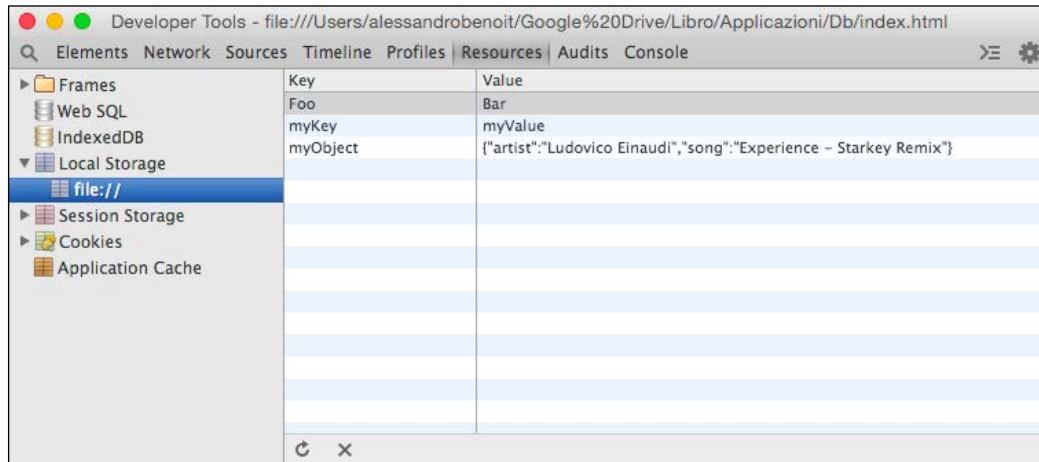
Note that `sessionStorage` has exactly the same properties and methods as those of `localStorage`.

As I stated before, `localStorage` only takes string as its value, so if you need to store more complex data, you have to *convert them first to JSON or base64*. The former is perfect to store normal JavaScript objects, whereas the latter is perfect for images and other kinds of files **Binary Large Objects (BLOBs)**.

In order to store a JavaScript object in `localStorage`, we can proceed as follows:

```
localStorage.myObject = JSON.stringify({  
  artist: 'Ludovico Einaudi',  
  song: 'Experience - Starkey Remix'  
});  
var myObject = JSON.parse(localStorage.myObject);  
console.log(myObject);
```

You can easily debug data stored in `localStorage` by opening **Developer Tools** and checking for the **Resources | Local Storage** tab, as shown in the following screenshot:



When you store a value in `localStorage`, a `storage` event is thrown in all the concurrent windows. So, open a new window and listen for the storage event as follows:

```
window.addEventListener('storage', function (e) {  
  console.log(e.key, e.newValue);  
});
```

You can actually communicate between the window emitting the event and the one listening (the storage event is not caught inside the very same window that's emitting it).

If you'd rather use higher-level APIs to store objects using web storage, you can rely on **StoreDB**, a very simple, *MongoDB-style* API that sits atop `localStorage`. You can install StoreDB by downloading it from GitHub (<https://github.com/djyde/StoreDB>) or with Bower:

```
$ bower install storedb
```

Then, include it in the head of your page with the following code:

```
<script type="text/javascript" src="bower_components/storedb/  
  /storedb.js"></script>
```

To store an object with StoreDB, you can simply type:

```
storedb('players').insert({
  name: 'Lucas',
  sex: 'male',
  score: 24
});
```

Then, to retrieve it, you can proceed as follows:

```
var players = storedb('players').find({name:'Lucas'});
console.log(players); // Array of players with the name Lucas
```

You can update values using MongoDB-style operators, such as `$inc` (increase), `$set` (replace), or `$push` (append):

```
// Add 10 to the score of the players named Randy
storedb('players').update(
  { 'name': 'Randy' },
  { '$inc': { 'score': 10 } }
);
```

Eventually, remove the data with the following code:

```
// Remove players named Randy
storedb('players').remove({ 'name': 'Randy' });
```

Using `find()` or `remove()` with no options will find or delete, respectively, all the objects within the given key. One last thing to note is that StoreDB also provides a callback function to handle errors. The result of each transaction will not be returned by the method itself but only through the callback (if you set it that way). Let's see an example:

```
var outcome = storedb('players').find({name:'Lucas'}, function (err,
result) {
  if(err) throw err;
  console.log(result); // Array of players named Lucas
});
console.log(outcome); // null
```

As illustrated, StoreDB is a very small project and has obviously the same flaws as `localStorage`, but it comes in really handy to store and find objects in small projects.

Web SQL Database

Web SQL Database is a specification based around *SQLite* that brings SQL to the client side. It has surely on its side the ease of the **SQL language** (besides select, insert, update and delete, you can do joins, inner selects, and so on) and the ability to be queried *asynchronously*, but it's also pretty verbose and **currently deprecated by W3C** as its implementation standards are open to debate. However, as it has not yet been discontinued, and it will probably not be for the time being, you're free to use it on your NW.js projects.

In order to play with Web SQL Database, you have to first create and open a database:

```
var dbName = 'myDb',
    dbVersion = '1.0',
    dbDisplayName = 'My Database',
    dbSize = 1024 * 1024;

var Database = openDatabase(dbName, dbVersion, dbDisplayName,
    dbSize, function(Database) {
    console.log("Database open");
});
```

The database connection is created on the fly and a Database object is returned in order to start a transaction.



In order to handle database versioning, use Database.changeVersion().

Transactions are containers for one or more SQL statements that are run sequentially. There are two types of transactions, `transaction()` and `readTransaction()`; the first one allows read/write operations, while the second allows only read operations, but it's faster. Both of them take two callbacks as arguments:

```
Database.transaction(SQLTransactionCallback,
    SQLTransactionErrorCallback);
function SQLTransactionCallback(SQLTransaction) {
    SQLTransaction.executeSql(...);
    SQLTransaction.executeSql(...);
}
function SQLTransactionErrorCallback(SQLError) {
    console.warn(SQLError);
}
```

`SQLTransactionCallback` is the callback used to wrap SQL statements, while `SQLTransactionErrorCallback` is an error callback called if one or more SQL statements fail.

`SQLTransaction.executeSql()` runs a SQL statement and provides four arguments:

```
SQLTransaction.executeSql (
    sqlStatement,
    arguments,
    SQLStatementCallback,
    SQLStatementErrorCallback
);
function SQLStatementCallback (SQLTransaction, SQLResultSet) {
    // Do something with SQLResultSet
}
function SQLStatementErrorCallback (SQLTransaction, SQLError) {
    // Do something with SQLError
    return Boolean;
}
```

Here's a description of the single arguments:

- The `sqlStatement` string: This is the actual SQL query.
- The `arguments` array (optional): This replaces all the `?` characters inside the SQL query with the corresponding values in order *to avoid SQL injection*.
- The `SQLStatementCallback` function (optional): This is a success callback that runs if no errors occurs. It provides a `SQLResultSet` object, which can be used to retrieve the result of the SQL statement.
- The `SQLStatementErrorCallback` function (optional): This is an error callback. It provides a `SQLError` object, which contains error code and messages.

Let's check the `transaction()` life cycle:

1. Open a new SQL transaction to the database. If an error occurs, go to step 5.
2. Run each SQL statement sequentially. If an error occurs and the given statement provides `SQLStatementErrorCallback`, go to step 4; otherwise, go to step 5.
3. Commit the SQL statements. If no errors occur, stop here; otherwise, go to step 5.

4. Check for returned value of `SQLStatementErrorCallback`; if the returned value is `true`, go to step 5; if `false`, handle the error and keep running the remaining statements through step 3.
5. Call `SQLTransactionErrorCallback` and roll back all the transactions.

It took me some time to figure out how commits, rollbacks, and errors were working inside a transaction. I hope I made it clear, but if I didn't, I'm sure the following example will clear it up:

```
var db = openDatabase('myDb', '1.0', 'My Database', 1024*1024);

db.transaction(function (t) {

    // Create articles TABLE
    t.executeSql("CREATE TABLE IF NOT EXISTS articles (id INTEGER
        PRIMARY KEY, title TEXT, content TEXT)");

    // Seed data into the articles TABLE
    var data = ['My test article title', 'My test article body'];
    t.executeSql(
        "INSERT INTO articles(title, content) VALUES (?, ?)", data
    );

    // Log the operation into the mylogs TABLE
    // which I assume already exists
    var message = ['articles Table created and seeded'];
    t.executeSql(
        "INSERT INTO mylogs(messages) VALUES (?)", message
    );

}, function(SQLError) {
    console.warn(SQLError.message);
});
```

In the preceding example, we have three statements. We first create an `articles` table, we seed some data into it, and eventually log a message in the `mylogs` table, which we assume already exists. What do you think would happen if our assumption was wrong and the `mylogs` table didn't exist? We would get a **no such table: mylogs** error from the `SQLTransaction` error callback and all the others statements would be rolled back.

Let's review the third statement:

```
var message = ['articles Table created and seeded'],
    query = "INSERT INTO mylogs(messages) VALUES (?)";
t.executeSql(query, message, null, function (t, SQLError) {
    console.warn('Could not log operation', SQLError.message);
    return false;
});
```

As illustrated, we're properly handling the error and making the error callback return `false`. Rerunning our code, we will still get the **no such table: mylogs** error, but this time, it will be fired by the statement itself, so all the others statements will be committed correctly.

Just as with web storage, you can check for WebSQL data by opening **Developer Tools** and clicking on **Resources | Web SQL**:



Once we have populated our database, we can proceed to read some data from it:

```
db.transaction(function (t) {
    t.executeSql("SELECT * FROM articles", [], function (tr,
        SQLResultSet) {
        var data = [], row, rowNumber = SQLResultSet.rows.length;
        for (var i = 0; i < rowNumber; i++) {
            row = SQLResultSet.rows.item(i);
            data.push(row);
        }
        console.log(data);
    });
});
```

```
    }, function(SQLError) {  
        console.warn(SQLError);  
    });
```

As illustrated, we make use of the `SQLResultSet` object. `SQLResultSet.rows` has a `length` property but doesn't behave like an array, so you need to loop it through using `SQLResultSet.rows.item(i)`. In our case, we're pushing the resulting row objects in a data array that should look something like the following:

```
[  
  {  
    "id":1,  
    "title":"My test article title",  
    "content":"My test article content"  
  }  
]
```

You can find more information about WebSQL specifications on the W3C website at <http://www.w3.org/TR/webdatabase/>.

IndexedDB

IndexedDB is a transactional database system thought to be *the NoSQL successor of WebSQL* and works by storing key-value pairs. The values can be complex JavaScript objects indexed by one or more object properties, which provide an efficient way to search a large set of items and properly handle multiple values per key. All the operations performed are done *asynchronously*, so they won't freeze the application. Furthermore, as there are *no storage limits*, IndexedDB can be used to handle a large amount of data. The only flaw could be seen in the lack of relational databases, but that's actually the point of **object-oriented database management systems**.

In order to open a database, we can proceed in the following manner:

```
var dbName = 'DatabaseName',  
    dbVersion = 1; // Int  
var request = indexedDB.open(dbName, dbVersion);
```

The `open()` method returns an `IDBOpenDBRequest` object, which inherits from `IDBRequest`. The `IDBRequest` interface provides means to access results of asynchronous requests to databases and exposes two event handlers:

```
request.onerror = function(event) {  
    // Do something with this.error.message!  
};  
request.onsuccess = function(event) {
```

```
// Do something with this.result!
};
```

The `onerror` event handler is fired when something goes wrong (for example, when the database versions mismatch), while `onsuccess` is called when the request succeeds. `IDBOpenDBRequest` also has an `onupgradeneeded` event handler, which will trigger before any other event when a new database is created or the version of an existing one is changed.

The `onupgradeneeded` event handler is the ideal place to put all the code to define our **database schema**:

```
request.onupgradeneeded = function(event) {
  var IDBDatabase = this.result;
  // Create an objectStore for this database
  var objectStoreName = 'MyObjectStore',
      objectStoreOptions = {
        keyPath: null,
        autoIncrement: false
      };
  var IDBObjectStore = IDBDatabase.createObjectStore
    (objectStoreName, objectStoreOptions);
};
```

In the preceding code snippet, we get the `IDBDatabase` object and then create an `IDBObjectStore` object named `MyObjectStore`. The `autoIncrement` option defines whether the main key should automatically increase when you add a new item and `keyPath` allows you to assign one of the properties of the stored objects as the main key (for example, `keyPath: SocialSecurityNumber`).



You can access the *database version* using `event.oldVersion` and `event.newVersion` inside `onupgradeneeded` in order to properly handle schema revisions. `IDBObjectStore.deleteIndex` allows you to remove an index, while `IDBDatabase.deleteObjectStore` allows you to delete `IDBObjectStore`.

Let's put together all we have learned:

```
var request = indexedDB.open('AppDB', 1);
request.onerror = function() {
  console.warn('Database error: ' + this.error.message);
};
request.onupgradeneeded = function(event) {
  var db = this.result;
```

```
var customers = db.createObjectStore('customers',
  {autoIncrement: true});
// Create extra indexes
customers.createIndex('by_name', 'name');
customers.createIndex('by_email', 'email', { unique: true });
};
request.onsuccess = function() {
  var db = this.result;
  // Do something once the db is ready
};
```

In the preceding example, we've added two extra indices in order to be able to search customers `IDBObjectStore` by name and e-mail. The `IDBObjectStore.createIndex()` method takes the name of the index, the key path (or an array of key paths) of the object property to be indexed and an object containing two extra options: the `unique` flag prevents the insertion of multiple objects sharing the same property value, while the `multiEntry` flag enables to map a single property of the array type to multiple single keys in the same index.

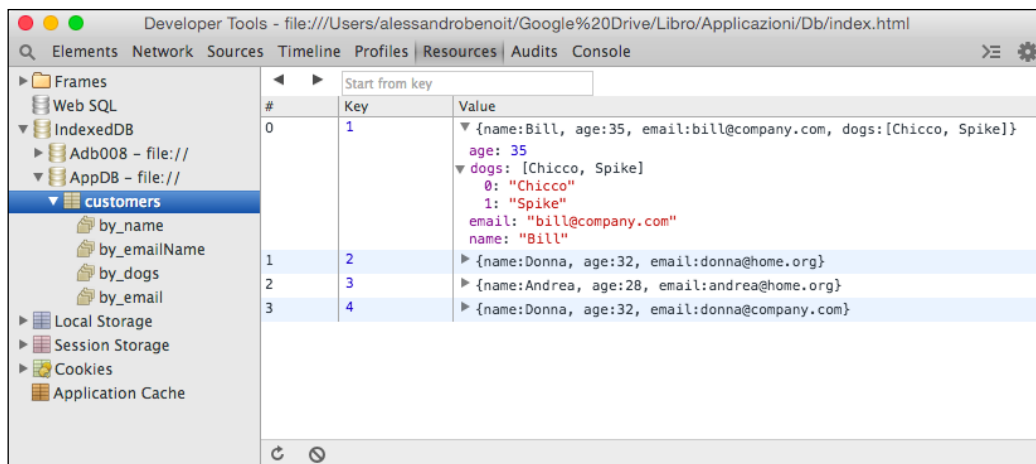
Once we have created our database and set its schema, we're ready to insert some data into it:

```
request.onsuccess = function() {
  var db = this.result;
  // Create a transaction
  var transaction = db.transaction(['customers'], 'readwrite');
  // Retrieve the IDBObjectStore
  var customers = transaction.objectStore('customers');
  // Add a new customer
  var request = customers.add({
    name: 'Marge Griffin',
    email: 'marge@example.com',
    phone: '+390541234567'
  });
  request.onsuccess = function () {
    // Output the Key of the new Object
    var ObjectKey = this.result;
    console.log(ObjectKey);
  };
};
```

 IndexedDB also supports storing **BLOB objects** without cumbersome conversions; check the example in the next section.

In the preceding simple example, we added a new object into `customers` `IDBObjectStore`. In order to do so, we created `IDBTransaction`, retrieved `IDBObjectStore`, and finally added the new object. The `IDBObjectStore.add()` method returns an `IDBRequest` object, the same one that the `Database` object inherited from. Eventually, you'll be able to retrieve the key of the newly added object by calling the `onsuccess` event handler on `IDBRequest`.

You can check whether the object has actually been added by opening **Developer Tools** on the **Resources** | **IndexedDB** tab, as shown in the following screenshot:



One more thing to consider is that the transaction supports three modes:

- `readonly`: This allows multiple concurrent transactions
- `readwrite`: Here, multiple concurrent transactions are not allowed and object stores and indexes can't be added or removed.
- `versionchange`: This is similar to a `readwrite` transaction, but it can create and remove object stores and indices.

What's missing here is some kind of **error handling**. If you think about running the same request twice, it would result in **duplicate index error** as the same e-mail would be used for more than one item. Much like in `WebSQL`, errors can be handled either on single requests or on full transactions, which may contain more requests.

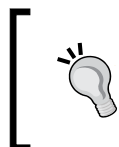
So, by default, *if a single request fails, the full transaction will be aborted*, rolling back all the requests within the transaction. Using `event.preventDefault()` on the single-request `onerror` event handler will disable the default behavior, letting you handle each request error individually. But a few lines of code speak louder than a million words:

```
var transaction = db.transaction(["customers"], "readwrite");
var customers = transaction.objectStore("customers");
// Add a new customer twice
for (var i = 2; i > 0; i--) {
    customers.add({
        name: 'Homer Griffin',
        email: 'homer@example.com',
        phone: '+390541234567'
    });
}
transaction.onabort = function (event) {
    console.warn(event.target.error.message);
};
```

In this case, as we are trying to insert two customers with the same e-mail, the transaction will be aborted and none of the requests will be committed. As a matter of fact, if you check `customers IDBObjectStore`, you won't find Homer Griffin at all. You can easily handle the exception as follows:

```
// Add a new customer twice
for (var i = 2; i > 0; i--) {
    customers.add({
        name: 'Homer Griffin',
        email: 'homer@example.com',
        phone: '+390541234567'
    }).onerror = function (event) {
        event.preventDefault();
        // Handle the exception with this.error.code
    };
}
transaction.onerror = function (event) {
    console.warn(
        'Transaction committed with errors',
        event.target.error.message
    );
};
```

This time, you will still get the error when the duplicated index key is found, but the first request will be committed correctly, and Homer Griffin will find a place in the customers' `IDBObjectStore`.



In order to make sure that a request has actually been committed, `IDBTransaction.onsuccess` is a much better place to check than `IDBRequest.onsuccess` because when the latter has been fired, something can still go wrong.

In order to update a record, we can replace the `IDBObjectStore.add()` method with the `put()` method, which acts much like `IDBObjectStore.add()` as the `put()` method will store a new value if a previous one is not found:

```
customerKey = 1; // Key for Marge Griffin
customers.put({
  name: 'Marge Griffin',
  email: 'newemail@example.com',
  phone: '+390541234567'
}, customerKey);
```

The `put()` method has an optional attribute to specify the key of the value to update. If the `IDBObjectStore` main key has been set with the `keyPath` option, you don't need to set the optional key as the assigned object property will be used. Let's say you had set `IDBObjectStore` as follows:

```
var cstmrs = db.createObjectStore('customers', {keyPath: phone});
```

Then, you could have updated the Marge object as follows:

```
cstmrs.put({
  name: 'Marge Griffin',
  email: 'newemail@example.com',
  phone: '+390541234567'
});
```

Eventually, we have a database with an object store and some data in it, so we can start doing some read operations. In order to get a single item by its key, you can proceed as follows:

```
var customerKey = 2; // Key for Homer Griffin customer
var request = customers.get(customerKey);
request.onsuccess = function() {
  if (!this.result) {
```



```
        console.warn('No customer found for key: ' + customerKey);
        return;
    }
    var customer = this.result;
    console.log(customer);
};
```

A single customer object corresponding to the given key will be returned by `this.result`. The previous statement can be shortened as follows:

```
customers.get(2).onsuccess = function () {
    if (!this.result) return;
    console.log(this.result);
};
```

If you need to get a value from an index, you can rewrite the preceding code as follows:

```
customers.index('by_email').get('homer@example.com')
    .onsuccess = function () {
        if (!this.result) return;
        console.log(this.result);
    };
};
```

To get multiple values, you have to iterate over the records using a **cursor**. A cursor comprises a range of records in either an index or an object store. So, to get all the data in your customers' store, use the following code:

```
var customersArray = [];
customers.openCursor().onsuccess = function () {
    var cursor = this.result;
    if (cursor) {
        customersArray.push(cursor.value);
        cursor.continue(); // Loop
    } else {
        // Do something with customers
        console.log(customersArray);
    }
};
```

To get all the stored objects from a given index, you can pass an `IDBKeyRange` object to the open cursor:

```
customers.index('by_name')
    .openCursor(IDBKeyRange.only('Homer Griffin'))
    .onsuccess = function (event) {
```

```
    if (!this.result) return;
    console.log(this.result.value);
    cursor.continue();
  };
```

IDBKeyRange defines a key range. You can use the `only()`, `lowerBound()`, `upperBound()`, and `bound()` methods to define the range.

In order to delete an item, you can rely on `IDBObjectStore.delete(key)`, as follows:

```
var customerKey = 2; // Key for Homer Griffin customer
customers.delete(customerKey).onsuccess = function () {
  console.log('Customer deleted');
};
```

As illustrated, IndexedDB is a really powerful tool to handle your data. I've tried to be as comprehensive as I could be, but there are still a few things you can learn about it by visiting the W3C specifications page at <http://www.w3.org/TR/IndexedDB/>.

If you find IndexedDB too verbose, you should take a look at **Dexie**, a minimalistic wrapper for IndexedDB (<http://www.dexie.org>). Instead, if you need a fast way to sync your application data with a server-side database you might consider **PouchDB**, which sits atop IndexedDB or WebSQL, and which provides two-way data synchronization with Apache **CouchDB** (<http://pouchdb.com>).

XMLHttpRequest and BLOBs

The one thing you should remember when dealing with `XMLHttpRequest` on NW.js applications is that local files, called through the `file://` or `app://` protocols, return 0 as the HTTP response.

```
var xhr = new XMLHttpRequest(), blob;
xhr.open('GET', 'myBlob.jpg', true);
xhr.responseType = 'blob';
xhr.addEventListener('load', function () {
  if (xhr.status !== 200) return;
  blob = xhr.response;
  // Do something with blob
}, false);
xhr.send();
```

In the preceding example, we try to get a **BLOB** (which, in our case, is an image), but the code will stop when checking for `xhr.status !== 200` as the returned status for local files will indeed be 0. I take this opportunity to show you how to store BLOBs in an **IndexedDB** object store:

```
var xhr = new XMLHttpRequest(), blob;
xhr.open('GET', 'maggieAvatar.jpg', true);
xhr.responseType = 'blob';
xhr.addEventListener('load', function () {
  maggieAvatar = xhr.response;
  customers.add({
    name: 'Maggie Griffin',
    email: 'maggie@example.com',
    phone: '+390541234567',
    avatar: maggieAvatar // Our blob
  });
});
xhr.send();
```

Then, in order to retrieve the stored image, we can proceed as follows:

```
<img id="myImg">
<script>
customers.index('by_name').get('Maggie Griffin')
  .onsuccess = function(event) {
    if (!this.result) return;
    document.getElementById('myImg').src = URL.createObjectURL
      (this.result.avatar);
  };
</script>
```

Handling media files

HTML5 APIs provide a simple way to handle media files such as **audio** and **video**. The API implementation is really easy:

```
<video width="400" controls>
  <source src="video.ogv" type="video/ogg">
</video>
<audio controls>
  <source src="audio.ogg" type="audio/ogg">
</audio>
```

Between the supported file formats, you can find `ogg`, `ogv`, and `wav`. The codecs included in the pre-built `ffmpegsumo` library are *Cotheora*, *vorbis*, *vp8*, *pcm_u8*, *pcm_s16le*, *pcm_s24le*, *pcm_f32le*, *pcm_s16be* and *pcm_s24be*.

As you have probably already figured out, you're not allowed to play `mp3` and `mpeg4` files natively as their codecs are proprietary. Unfortunately, there's no easy workaround for the issue. Someone has successfully used the media libraries from **Chrome**, but I couldn't replicate the process on NW.js 0.11.5. The best way to give support for these codecs to your application is to compile `ffmpeg` with the corresponding options, but bear in mind that the MP3 and H.264 codecs are licensed under the *GPL license* in `ffmpeg`, which means that any derived work must be released under the same license. You can find a comprehensive guide on building `ffmpeg` on the NW.js GitHub Wiki page.

Shedding some light on security issues

When developing a desktop application, most of the *assets usually come from trusted sources*, so in NW.js, many of the security precautions implemented by Chromium have been disabled. However, we must distinguish between **Node frames** and **Normal frames**. The first kind of frames are the ones we have dealt with in previous chapters, while the latter kind are normal browser frames, which act much like Chrome frames.

With regard to security issues, Node frames are allowed to:

- Access `require`, `global`, `process`, `Buffer`, and `root` from `Node.js`
- Access other frames by skipping the **cross-domain** security checks
- Ignore the `X-Frame-Options` headers for child frames

By default, the following resources will be handled with Node frames:

- Local resources
- App protocol resources, for example, `app://myApp/index.html` (for more information on this, refer to *Chapter 6, Packaging Your Application for Distribution*)
- Remote resources specified in the `node-remote` option in your manifest file, for example, `"node-remote": "*.google.com, *.twitter.com"`

The following resources are handled with Normal frames:

- Remote resources
- Windows opened with Native UI APIs, which set the `new-instance` flag to `true` and `nodejs` to `false`. Here's an example:

```
var gui = require('nw.gui');
gui.Window.open('localFile.html', {
  'new-instance': true,
  'nodejs': false
});
```

Every time you deal with untrusted code and content, you should stick to Normal frames and iframes. In this regard, **iframes** should be handled with particular attention. In order to load untrusted resources inside an iframe, you must specify the `nwdisable` and `nwfaketop` attributes as follows:

```
<iframe src="http://www.google.com" style="width: 80%;" nwdisable
  nwfaketop>
```

You can also specify an optional user agent with the `nwUserAgent` attribute:

```
<iframe src="http://www.google.com" style="width: 80%;" nwdisable
  nwUserAgent="Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_0) AppleWebKit/537.36 (KHTML, like Gecko)
  Chrome/39.0.2171.95 Safari/537.36">
```

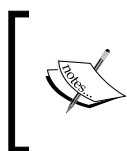
A very, very simple in-app browser implementation might look like this:

```
<input id="url" type="text" placeholder="Insert the website
  address">
<iframe id="browserFrame" src="http://www.google.com"
  style="width: 100%;height:400px;" nwdisable nwfaketop
  nwUserAgent="Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_0)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95
  Safari/537.36"></iframe>
<script type="text/javascript">
  var addressBar = document.getElementById('url');
  addressBar.addEventListener('keydown', function(e) {
    if (e.keyCode == 13)
      document.getElementById('browserFrame').src = addressBar
        .value;
  });
</script>
```

In this case, you might also consider enabling external plugins (such as Adobe Flash Player), Java support, and page caching. In order to do so, you have to edit your manifest file as follows:

```
{
  "name": "inapp-browser",
  "main": "index.html",
  "webkit": {
    "plugin": true,
    "java": true,
    "page-cache": true
  }
}
```

Remember to give your application a chance to clear the cache at runtime with `gui.App.clearCache()`. Page caching in NW.js is disabled by default.



I have to disprove myself as, doing a test on NW.js 0.11.5, I've found out that whether page-cache is set to `true` or `false`, whenever I visit a remote website, some data is stored in the cache folder within the `dataPath` folder. I guess it's some kind of bug, so I've reported it.

The Web Notifications API

Notifications are small popups that are usually shown in the top-right corner of the screen that the operating system implements in order to *share important events* with the user. Each notification features a title, some descriptive text, and, optionally, a small thumbnail/icon; more notifications can be stacked.

I was really tempted to insert this argument in the **Native UI APIs** chapter as it shares the same goals, but Notifications APIs, although relatively young, are actually part of **HTML5** specifications.

The code to implement web notifications inside your NW.js application is pretty straightforward:

```
var notification = new Notification('Notification Title', {
  icon: "icon32.png",
  body: "Here is the notification text"
});
notification.onclick = function () {
  console.log("Notification clicked");
};
```

```
notification.onshow = function () {  
  console.log('Show');  
  // Close the Notification after 1 second  
  setTimeout(function() {notification.close();}, 1000);  
};
```

You can create a notification with `new Notification('Title', options)`, where options are the notification icon (32 x 32 pixels) and its body. Once the notification has been created, you can listen in for the `click` and `onshow` events in order to respond appropriately.

Web notifications are highly dependent on the operating system, and here, things get trickier as *each OS implements notifications in a different way*:

- On **Mac OS X**, web notifications are shown for a predetermined period of time, so neither the `Notification.onshow` event nor the `Notification.close()` method will work.
- On **Microsoft Windows** versions higher than 8, in order to show a notification, you have first to create an application shortcut on the Start screen, providing `AppUserModelID`. It looks harder than it is as you can easily set `AppUserModelID` from the manifest file using `{"app-id": "com.node.webkit.notification.test"}` and then create the shortcut with the built-in app API:

```
var gui = require('nw.gui');  
gui.App.createShortcut(process.env.APPDATA +  
  "\\Microsoft\\Windows\\Start Menu\\Programs\\NW.js.lnk");
```
- On **Linux**, neither the `Notification.onshow` event nor the `Notification.onclick` event nor for that matter the `Notification.close()` method works.

Summary

As you've probably noticed, I spent most of this chapter dealing with **data persistence solutions** since it is not only the trickiest argument, but also essential to develop real desktop applications.

Browser Web APIs are huge — probably ten books like this one wouldn't be enough to completely explore the topic. Fortunately, there's a lot of material online, so it shouldn't be too hard to dive deep into the argument.

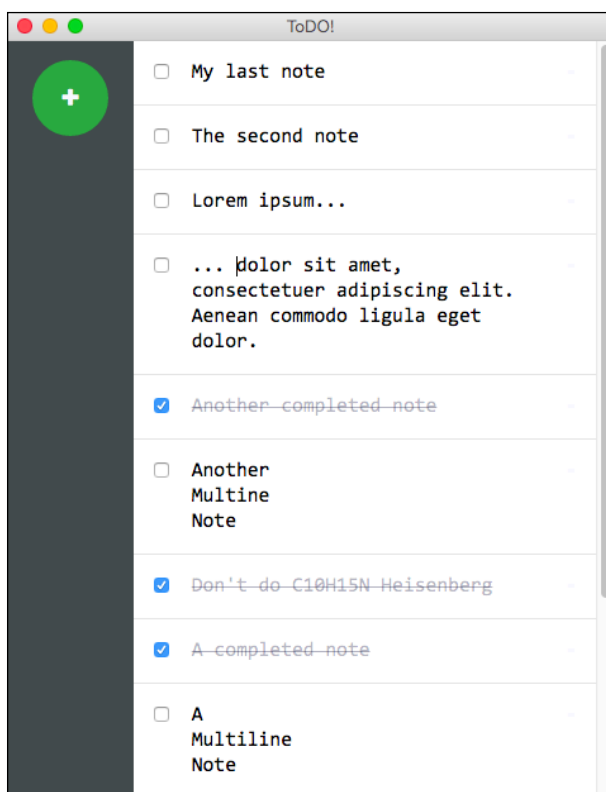
At this point, you've learned how to interact with the OS GUI, how to leverage Node.js low-level functions, and how to play nice with Browser Web APIs. All that is left to do is to fit together all the pieces in order to realize a **modern desktop application**, which is exactly what we're going to do in the next chapter.

Let's code...

5

Let's Put It All Together

Finally, we're ready to put into practice what we've learned until now. In this chapter, we're going to *develop a full, working NW.js desktop application from scratch*. We will do it step by step, starting from some **styling advice**, proceeding by implementing the **basic functionalities**, and eventually integrating them to create a proper **desktop GUI** thanks to Native UI APIs. Here's how our application will look like:



I apologize for the lack of imagination, but I thought that the most suitable example was the classic **ToDo list application**. However, in order to make it interesting, I added the ability to *synchronize the database remotely via PouchDB* and a simple option panel to better understand how to work with multiple windows. Moreover, the application will be ready to run on Mac OS X, Microsoft Windows, and Linux.



Some notes about the programming style adopted in this chapter

The programming style adopted in this chapter is probably much different from the one I would have used in a production application. I hope you won't reproach me for this, but in order for it to be understandable by programmers with different backgrounds, I had to make a choice. So I decided to use **jQuery** to manipulate the DOM instead of an MVC Web application framework, and I **namespaced** everything to keep the learning path as linear as possible. Moreover, since most of the readers of the book are already skilled io.js/Node.js developers, I decided to focus on Browser Web APIs and Native UI APIs.

Let's get started!

Here we are. The first thing I consider when developing a new project is usually the application folder structure:

```
myApp/  
  > css/  
    -- app.css  
    -- options.css  
  > js/  
    -- nativeui.js  
    -- helpers.js  
    -- main.js  
    -- options.js  
    -- todos.js  
  > node_modules/  
  > vendor/  
    -- jquery.2.1.3.min.js  
    -- pouchdb-3.2.1.min.js  
  > views/  
    -- app.html  
    -- options.html  
    -- todo.html  
  - icon16.png
```

- index.html
- options.html
- package.json

As illustrated, before starting with the actual programming, we need to install some resources. So let's create the `myApp` folder and all the subfolders and then create a `package.json` file in the root:

```
{
  "name": "ToDo!",
  "main": "index.html",
  "window": {
    "show": true,
    "icon": "icon16.png",
    "min_width": 480,
    "min_height": 600,
    "width": 480,
    "height": 600,
    "toolbar": false
  },
  "dependencies": {
    "swig": "1.4.2"
  }
}
```

What we are doing here is pretty simple; we've set most of the data of the application, including the Main window size and icon, which will be shown on some OS beside the application title bar, and eventually added the **Swig template engine** as a dependency. In order to install the dependencies, just run `npm install` in the terminal within the application's root directory.

Now you will need a 16 x 16 pixel icon (you can find one under an MIT license on iconfinder.com); place it in the root directory with the name `icon16.png`.

We'll be using **jQuery 2.1.x** for DOM manipulation and **PouchDB 3.2.x** as the database; you can download the latest version of each at the following web addresses:

- <http://jquery.com/download/>
- <http://pouchdb.com/>

Once you have downloaded the packages, you can put the minified files in the vendor folder.

A matter of style

Once we have our folder structure ready, we can proceed to create the style sheets. Inside the `css` folder, create an `app.css` file for the Main window and an `options.css` file for the Options window. Let's see the first one:

`app.css`:

```
/* Reset */
html, body, #mainView {
  width: 100%; height: 100%; margin: 0; padding: 0;
}
/* Custom behaviors */
html * {
  -webkit-user-select: none; box-sizing: border-box;
}
/* mainView container */
#mainView {
  display: flex; border-top: 1px solid #c7c7c7;
}
/* Right sidebar */
#sidebar {
  flex: 0 0 100px;
  padding: 15px 0; text-align: center;
  background: #414A4C; color: #f8f8f8;
}
/* Todo container */
#todoListContainer {
  flex: 1;
  background: #F8F8FF; overflow: auto;
}
/* Button styling */
.button {
  font-family: sans-serif; font-weight: 800;
  display: inline-block; border-radius: 100%;
  color: #F8F8FF; text-align: center; text-decoration: none;
  transition: background-color 0.2s linear;
}
/* Add button */
.add-button {
  background: #28a93f;
  height: 60px; width: 60px;
  line-height: 60px; font-size: 26px;
```

```
}
.add-button:hover {
  background-color: #46BA5A;
}
/* Delete button */
.delete-button-container {
  flex: 0 0 30px; text-align: right;
}
.delete-button {
  background: rgba(0,0,0,0.0);
  height: 20px; width: 20px;
  line-height: 18px; font-size: 18px;
}
/* ToDo List */
#todoList {
  list-style: none;
  padding: 0;margin: 0;
}
#todoList li {
  display: flex;
  border-bottom: 1px solid rgba(0,0,0,0.1);
  padding: 15px 10px; background: #F8F8FF;
}
#todoList li:hover .delete-button {
  background: rgba(0,0,0,0.1);
}
#todoList li:hover .delete-button:hover {
  background: #d62700;
}
#todoList input {
  flex: 0 0 30px;
}
/* ToDo Content */
.todo-content {
  flex: 1;
  line-height: 20px; height: 20px;
  padding: 0; font-size: 16px; color: #333;
  font-family: Consolas, monospace;
  overflow: hidden; resize: none;
  border: none; background: transparent;
}
.todo-content[disabled] {
```

```
    text-decoration: line-through; color: rgba(0,0,50,0.3);
  }
  .todo-content:focus {
    color: #000; outline: 0px solid transparent;
  }
  /* Sync overlay */
  #syncing {
    font-family: Consolas, monospace;
    display: block; position: fixed;
    top: 0;right: 0;bottom: 0;left: 0;
    background: rgba(0,0,0,0.8);
    text-align: center; color: #18E713;
  }
  #syncing > span {
    margin-top: 60px; display: block; font-size: 15px;
  }
  #syncing > span > span {
    -webkit-animation: blink 1s linear infinite;
  }
  /* Animation */
  @-webkit-keyframes blink {
    from { opacity: 1.0; } to { opacity: 0.0; }
  }
```

You can simply copy and paste the preceding code in order to proceed, but there are a few things you should note:

- In order to style an NW.js application, **flex** is probably one of the best CSS properties as it defines a flexible layout inside a container
- Applying `-webkit-user-select: none;` to all elements will prevent unexpected text selection on application elements. You can enable it with `-webkit-user-select: all;` when needed
- As stated in *Chapter 4, Data Persistence Solutions and Other Browser Web APIs*, there's no need to define any browser prefix other than `-webkit-`
- In our example, I have provided a basic styling reset, but you'd better use a more comprehensive one like the one by **Eric Meyer**

And here's the Options page style; I kept it as simple as possible:

options.css:

```
/* Reset */
html, body {
```

```
    width: 100%; height: 100%;
    margin: 0; padding: 0;
}
/* Custom behaviors */
html * {
    box-sizing: border-box; -webkit-user-select: none;
}
/* Body */
body {
    background: #f4f4f4; font-family: sans-serif;
    padding: 20px; border-top: 1px solid #ccc;
}
/* Labels */
label {
    display: block; border-bottom: 1px solid #eee;
    margin-bottom: 15px; color: #151515;
}
/* Inputs */
input {
    margin: 5px 0 15px 0;
}
input[type=text] {
    padding: 5px; width: 100%;
}
input[type=color] {
    display: inline-block; margin-left: 20px;
}
/* Buttons */
.buttons {
    position: absolute; bottom: 20px; right: 20px;
}
```

The HTML5 skeleton

Let's now create the HTML skeleton of the `index.html` and `options.html` pages. They will share the same structure but include different styles and JavaScript files:

`index.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>ToDo!</title>
```



```
<link rel="stylesheet" href="css/app.css"></link>
<script type="text/javascript" src="vendor/jquery.2.1.3.
  min.js"></script>
<script type="text/javascript" src="vendor/pouchdb-3.2.1.
  min.js"></script>
<script type="text/javascript" src="js/main.js"></script>
<script type="text/javascript" src="js/todos.js"></script>
<script type="text/javascript" src="js/nativeui.js"></script>
<script type="text/javascript" src="js/helpers.js"></script>
</head>
<body>
  <div id="mainView"></div>
</body>
</html>
```

In `options.html`, you can replicate the same code just by replacing all the head content with the following content:

```
<title>ToDo! &raquo; Options</title>
<link rel="stylesheet" type="text/css" href="css/options.css"></link>
<script type="text/javascript" src="vendor/jquery.2.1.3.min.js"></script>
<script type="text/javascript" src="js/options.js"></script>
<script type="text/javascript" src="js/helpers.js"></script>
```

There's not much to explain about the preceding code; we've defined all the needed files and added a `#mainView` div that we will be using to load templates with Swig.

Let's dive deep into the application logic

In the last few pages, we have set the manifest file and created the style sheets and the HTML skeleton of the different views. Now, we can finally dive deep into the actual application logic. The `main.js` file is nothing more than a wrapper that defines the namespace and initializes the different parts of the application:

```
var myApp = {};
myApp.init = function () {
  // Instance the DB
  myApp.db = new PouchDB('todos');
  // Init the todo App
  myApp.todos.init();
};
$(window).load(myApp.init);
```

As illustrated, we are initializing the application on window load in order to make sure that everything is loaded before declaring any logics. In the `myApp.init()` function, we're instantiating a PouchDB database object, which we'll be using later to access and store our to-do lists.

As you can find on its website, **PouchDB** is an open source JavaScript database inspired by Apache CouchDB that is designed to run well within the browser. Its peculiarity is that it sits on top of **indexedDB** to store data on the browser, and it's incredibly easy to sync it with **CouchDB**, so you don't need to create a lot of extra logic to handle **two-way data replication**.

In order to debug the data stored with PouchDB, you can proceed as you would for indexedDB, opening the **Developer Tools** and clicking on **Resources | IndexedDB**.

However, keep in mind that the data might not look as you expected since, to implement proper synchronization, PouchDB automatically builds a lot of extra object stores and keep revisions of each stored object.

After initializing the database, we call the `init` function of the actual **ToDo** application. We'll see all of that in a minute, but first we need to populate the `js/helpers.js` file with a few functions that we'll be using globally in our application:

```
// Load Template
myApp.loadTemplate = function (view, data, target) {
  var swig = require('swig'),
      fileName = 'views/' + view + '.html';
  var template = swig.renderFile(fileName, data || {});
  if (target)
    return $(target).html(template);
  return template;
};

// Remove elements from the DOM after slideUp
myApp.removeFromGUI = function ($items) {
  $items.slideUp(200, function () {
    $(this).remove();
  });
};

// Autoresize textarea on add/delete content
myApp.resizeTextarea = function ($ta) {
  $ta.height(20).height($ta.prop("scrollHeight"));
};
```

We're loading three functions in the `myApp` namespace. The first one will help us in the tedious operation of loading templates with the Swig module. The `view` attribute is the name of the file inside the views folder without the HTML extension; the `data` attribute (optional) is an array of data to be parsed by Swig; the `target` attribute (optional) is the ID of the DOM element in which we'll load the resulting HTML. If the `latest` attribute is not set, the HTML template will be returned by the function itself.

We then have the `removeFromGUI()` function that slides up and removes from the DOM any jQuery object provided and `resizeTextarea()`, a simple hack that enables text areas to autoresize depending on their content. Once we're done with the helpers, we're ready to start coding the inner logics of our application. We will do it in two steps:

1. Implement the actual application layer.
2. Implement the NativeUI layer.

The application layer

The application layer is all the logic related to the functioning of the TODO list application. In this first step, we will indeed create a full, working application, but we will do this much like we would do with a normal web page (except for the availability of Node.js modules). Before writing even a single line of code, let's see which features will implement the application layer:

- Load all the to-dos from the database on start
- Add a new to-do when clicking on a given button
- Each to-do item in the DOM will provide a way to the following:
 - Set the to-do status (completed/uncompleted) and save
 - Edit the to-do content and save
 - Delete the to-do

There is then a set of features that will be triggered by the NativeUI layer but will result in an action inside the application layer:

- Show/hide completed to-dos
- Change the to-do list background color
- Sync the local database with the remote one
- Export the to-do list in the JSON format

Once you've understood what the application will do, we can keep moving to the fun part... the coding! As first we need to implement an HTML skeleton for the TODO application, let's open `views/app.html` and add the following:

```
<!-- Export file dialog -->
<input style="display:none;" id="fileDialog" type="file"
  accept=".json" nwsaveas="todos.json"/>
<!-- Left sidebar -->
<div id="sidebar">
  <a href="#" class="button add-button">+</a>
</div>
<!-- ToDo List -->
<div id="todoListContainer">
  <ul id="todoList"></ul>
</div>
<!-- Sync in progress -->
<div id="syncing" style="display: none"><span>SYNC IN PROGRESS
  <span>_</span></span></div>
```

Let's analyze the preceding code:

- The `#fileDialog` file input field will open the **Save As** dialog thanks to the `nwsaveas` attribute, as seen in *Chapter 2, NW.js Native UI APIs*
- The `#sidebar` div element contains `.add-button` that, once clicked, will add a new to-do item
- The `#todoList` ul element is the container for the to-do list items
- The `#syncing` div element is an overlay that will show **SYNC IN PROGRESS** while PouchDB is syncing; it is hidden by default

Once we have the application skeleton set, we can add some logic to `todos.js`. So, let's start writing the `init` function:

```
myApp.todos = {};
myApp.todos.init = function () {
  // Load main application view
  myApp.loadTemplate('app', null, '#mainView');
  // Change todos bg color
  myApp.todos.changeBackgroundColor(localStorage.notesColor ||
    '#F8F8FF');
  // Listen for events
  myApp.todos.listenForEvents();
  // Load data from DB
  myApp.todos.load();
}
```

```
};  
myApp.todos.listenForEvents = function () {  
  // Add new todo  
  $(' .add-button').click(function (e) {  
    e.preventDefault();  
    myApp.todos.addNew();  
  });  
};  
myApp.todos.load = function () {};
```

In the preceding piece of code, we created the `todos` namespace, and then, inside the `init` function, we started loading the app template with `myApp.loadTemplate()`, which will load `views/app.html` inside the `#mainView` div.

When initializing the TODO application, we must also set the background color of the to-do list, so we call `myApp.todos.changeBackgroundColor()` passing the `localStorage.notesColor` option (if set) or the default color `#F8F8FF`. Let's then add the function at the end of the `todos.js` file:

```
myApp.todos.changeBackgroundColor = function (color) {  
  var $customStyle = $('<style/>', {  
    html: '#todoListContainer , #todoList li {background:'+ color  
      +';}',  
    id: 'customStyle'  
  });  
  if ( $('#customStyle').length > 0 ){  
    $('#customStyle').replaceWith($customStyle);  
  } else {  
    $customStyle.appendTo('head');  
  }  
};
```

As illustrated in the preceding code, `changeBackgroundColor()` will simply add a style tag to the head of the document, setting the to-do list's background color.

Adding a new task

Now, we can start listening for events and, more specifically, for the `.add-button` click event that will call `myApp.todos.addNew()`. Let's then create the function as follows and add it right after the rest of the code:

```
myApp.todos.addNew = function () {  
  var todo = {  
    _id: new Date().toISOString(),
```

```

        content: '',
        checked: false
    };
    myApp.db.put(todo, function callback(err, result) {
        if (err) return console.warn(err);
        todo._rev = result.rev;
        myApp.todos.render(todo, true);
    });
};

```

In order to create a new to-do item, we create an empty `todo` object by providing an `_id` property, which is essential for PouchDB APIs to properly insert a new item into the database. We could provide any kind of random string as `_id`, but using the date, we'll be able to sort the results chronologically with a much lower computational expense.

Once we've created the `todo` object, we push it to the database using `myApp.db.put()` and then, if no error occurs, we can render it to the screen with `myApp.todos.render()`. In the while we took to assign the `_rev` property to the `todo` object, the `_rev` field makes possible database replication, and this is essential for update/delete operations to work.

Let's add the following just after the previous piece of code:

```

myApp.todos.render = function (todo, focus) {
    var todoTemplate = myApp.loadTemplate('todo', todo),
        $todo = $(todoTemplate).prependTo('#todoList').hide().
            slideDown(300),
        $content = $todo.find('.todo-content'),
        $check = $todo.find('.todo-check');
    myApp.resizeTextarea($content);
    // Focus on the created todo
    if (focus) $content.focus();
    // -- Add events here --
};

```

The `myApp.todos.render()` method takes two arguments. The first is the `todo` object that, at this point, should look something like this:

```

{
    _id: "2015-01-24T18:43:36.234Z",
    _rev: "114-61d532da6f6d5bdbedabd39416d0c9d4",
    content: "",
    checked: false
}

```

The `focus` argument is a Boolean that determines whether the to-do content must be focused when called, which in this case is `true`.

In the first row of code, we load the template for a single to-do, so let's edit `views/todo.html` as follows:

```
<li>
  <input class="todo-check" type="checkbox" {% if checked
    %}checked{% endif %}>
  <textarea class="todo-content" {% if checked %}disabled{% endif
    %}>{{content}}</textarea>
  <div class="delete-button-container">
    <a href="#" class="button delete-button">-</a>
  </div>
</li>
```



If you're not used to Swig templates, I suggest that you visit its website <http://paularmstrong.github.io/swig/>.

Once we've parsed the template, we prepend it to `#todolist ul`, create a few jQuery objects referring to the to-do DOM elements, and eventually focus on its content and resize the text area. Even when the to-do is shown, we are not done as we still have to implement a series of actions to be triggered when we click on the delete-button link or change its content/status. Let's add the following events just after the `-- Add events here - comment`:

```
// On check/uncheck item
$check.on('change', function () {
  todo.checked = $(this).is(':checked');
  myApp.todos.update(todo);
  // Disable on check
  if (todo.checked){
    if (localStorage.hideCompleted === 'true')
      myApp.removeFromGUI($todo);
    else
      $content.attr('disabled', true);
  } else {
    $content.removeAttr('disabled');
  }
});
```

When the checkbox status is changed, there are two consequences. First, the database version of the `todo` object must be updated, and we do this with `myApp.todos.update()`. Then, we must check the option that determines whether completed to-do items must be hidden in order to decide whether the task must be removed from the view once completed.

The next thing we must listen for is any change to the to-do content:

```
// On content change
var timer;
$content.on('input', function () {
  myApp.resizeTextarea($content);
  if (typeof timer !== 'undefined') clearTimeout(timer);
  timer = setTimeout(function() {
    todo.content = $content.val();
    myApp.todos.update(todo);
  }, 1000);
});
```

When we input something into the text area, the database must be updated, but, in order not to overload it, we've set a timer that will save changes only when not typing for 1 second. As illustrated, we're also calling the `resizeTextarea()` helper that will resize the text area in real time when some content is added or removed. As you have seen both `$content` and `$check` change, events will call `myApp.todos.update()`, which we've still not implemented, so let's create it and add it after `myApp.todos.render()`:

```
myApp.todos.update = function (todo) {
  myApp.db.put(todo, function callback(err, result) {
    if (err) return console.warn(err);
    todo._rev = result.rev;
  });
};
```

As illustrated, `myApp.todos.update()` is just a wrapper for the PouchDB `put()` API. One thing you should note is that in the callback, we're reassigning the `todo._rev` property as it changes at every update.

We have still to enable the user to delete a to-do item by clicking on the `.delete-button`. Add the following piece of code under the other events inside `myApp.todos.render()`:

```
// On delete item
$todo.find('.delete-button').on('click', function (e) {
  e.preventDefault();
```



```
myApp.removeFromGUI($todo);  
myApp.db.remove(todo);  
});
```

We've already seen the `myApp.removeFromGUI()` helper and `myApp.db.remove()` is pretty much self-explanatory.

Loading all the tasks

If we try to run our application, at this point, most of the to-do features would work; however, there's still something missing. First, we need to load all the items into the view when the application is opened, so let's edit the `myApp.todos.load()` function that should already have been defined and called by `myApp.todos.init()`:

```
myApp.todos.load = function () {  
  myApp.removeFromGUI($('#todoList').find('li'));  
  if (localStorage.hideCompleted === 'true')  
    myApp.todos.loadUncompleted();  
  else  
    myApp.todos.loadAll();  
};
```

The `myApp.todos.load()` method first has to clean up any items previously added to the to-do list container; then, it has two possibilities. It may load all the tasks previously saved or only those tasks whose status is still "not completed".

In the first case, it would call the `myApp.todos.loadAll()` function, which you can add right after `myApp.todos.load()`:

```
myApp.todos.loadAll = function () {  
  myApp.db.allDocs({include_docs: true}, function(err, resp) {  
    resp.rows.forEach(function (item) {  
      myApp.todos.render(item.doc);  
    });  
  });  
};
```

The PouchDB `allDocs()` API returns all the stored objects asynchronously inside the `resp.rows` property. All we need to do is loop over it and render the items with `myApp.todos.render()` as we did when creating a new to-do.

I could have showed you the condition to show only uncompleted tasks right inside the loop, but I wanted to show you the difference between `PouchDB.allDocs()` and `PouchDB.query()`.

Let's see how the second one works in `myApp.todos.loadUncompleted()`:

```
myApp.todos.loadUncompleted = function () {
  function map (doc) {
    if (doc.checked === false)
      emit();
  }
  myApp.db.query(map, {include_docs : true}, function (err, resp)
  {
    resp.rows.forEach(function (item) {
      myApp.todos.render(item.doc);
    });
  });
};
```

The `query()` method takes a `map` function to establish whether a given item should be returned. You have probably noticed that in both cases, the `include_docs` option is set to `true`. That's actually really important; otherwise, only the `item` key would be returned, whereas in our case, the whole to-do object is needed.

Implementing export and sync features

Now, we need to implement the JSON export feature. The actual export will be triggered by the menu in the NativeUI layer; however, the feature itself belongs to the application layer, so let's add a new event inside the `myApp.todos.listenForEvents()` function:

```
// Export data
$('#fileDialog').on('change', function () {
  myApp.todos.export($(this).val());
  $(this).val(''); // Reset the value
});
```

When the value of the file input field inside `views/app.html` is changed, we call the `myApp.todos.export()` function, which we can add at the end of the `todos.js` file:

```
myApp.todos.export = function (path) {
  var fs = require('fs'),
      data = [],
      jsonData;
  myApp.db.allDocs({include_docs: true}, function(err, resp) {
    resp.rows.forEach(function (item) {
      var newItem = {
        content: item.doc.content,
```

```
        completed: item.doc.checked
      };
      data.push(newItem);
    });
    fs.writeFile(path, JSON.stringify(data, null, 2),
      function(err) {
        if(err) return alert(err);
      });
  });
};
```

Here, we're using the `fs` native Node.js module to write the result of `PouchDB.allDocs()` to the file path provided by the file dialog.

We're almost done with the application layer. All that is left to do is implement the two-way data replication with CouchDB.



I will not dive deep into how CouchDB works; you can easily implement your own CouchDB server (<http://couchdb.apache.org/>) or get a free one at <https://www.iriscouch.com/>. In both cases, just remember that if you get error 405 when trying to sync your database, it might depend on CORS, so you'll have to set them properly.

At the end of the `todos.js` file, add:

```
myApp.todos.sync = function () {
  if (!navigator.onLine)
    return alert('No internet connection available');
  $('#syncing').show();
  PouchDB.sync('todos', localStorage.couchDbUrl)
    .on('complete', function (info) {
      myApp.todos.load();
      $('#syncing').fadeOut();
    }).on('error', function (err) {
      alert(err);
      $('#syncing').fadeOut();
    });
};
```

In the first line of code, we check whether there's an Internet connection, and then we overshadow the application screen with the syncing overlay, which was previously added to `views/app.html`.

In order to start the synchronization, we call `PouchDB.sync()` by passing the name of the local database and the URL of the remote CouchDB—something of the `https://xxx.iriscouch.com/todo` kind. Once the two-way data replication is done, we listen for two possible events:

- On complete: Hide the overlay and reload the to-do list
- On error: Hide the overlay and alert

We're finally done with the application layer. Unfortunately, many features are still not accessible to the user as they must be triggered in order to work. In the next section, we will see how to implement context and window menus, option windows, and a few other features that are essential for our application to be usable and fully working around different platforms.

The NativeUI layer

As we have done with the application layer, before writing any line of code, we'd better take a look at which features we need the NativeUI layer to implement:

- Implement a Window menu
- Implement a Context menu on the to-dos content
- Restore the Main window position when opening the application
- Implement an Options window
- On exit, store the windows position, let the Options window save data and compact the database
- Open the application smoothly when everything has been loaded

First, we need to call `myApp.nativeUI.init()` from inside `myApp.init()` in `js/main.js`, and then we can open the `js/nativeui.js` file and add the following code:

```
myApp.nativeUI = {};
myApp.nativeUI.init = function () {
  // Instance nw.gui
  myApp.gui = require('nw.gui');
  // Globals
  myApp.mainWindow = myApp.gui.Window.get();
  myApp.name = myApp.gui.App.manifest.name;
  // -- Call the other functions here --
};
// -- Declare the other functions here --
```

In the first two rows, we performed the following actions:

- Created the `nativeUI` namespace
- Created the `myApp.nativeUI.init()` function, which was called by `myApp.init()`
- We created the following variables:
 - `myApp.gui`: This is the Native UI API's object
 - `myApp.mainWindow`: This is the Main window object
 - `myApp.name`: This is the application name as set in the manifest file

We'll use the preceding variables all around the NativeUI layer, so it was pointless to re-declare them at the time of need. Once we are done, we can start implementing the aforementioned features.

Implementing the Window menu

The window menu will implement the following submenus:

- **File**: This contains an **Exit** menu item (on Mac OS X, it's called the **App** menu and it implements a few other features)
- **Edit**: This contains **Cut**, **Copy**, **Paste**, and **Select All** (on Mac OS X, it implements a few other features)
- **Options**: The following are the items available under this:
 - **Options**: This will open the Options window
 - **Hide completed**: This is a checkbox item that will hide completed tasks
 - **Export**: This exports the to-do list to the JSON file
 - **Sync**: This manually triggers two-way data replication with the CouchDB DB set in the Options window, which, if not done, must result in this voice being disabled

To make the developer's life easier, we've seen that NW.js provides a `createMacBuiltin()` API that automatically creates Mac OS default menus. This approach, however, has a few flaws, the most evident of which is that there's no similar API on the others platform, so we'll have to manually build those menus for Microsoft Windows and Linux. Moreover, calling `createMacBuiltin()` creates a localized menu, so as our example application doesn't provide localization, on my MacBook I'll see the **App** and **Edit** menu in Italian and the **Options** menu in English. That's surely not what we expect from a native application, but unfortunately, in our case, we will have to live with it.

However, when building your own application, you can consider two possible approaches:

- Localize your application using `window.navigator.language`
- Implement the Mac OS X menu from scratch

In order to implement the Window menu, we first have to add a call to `myApp.nativeUI.createWindowMenu()` inside `myApp.nativeUI.init()`. Once you've done that, we can add the function right after it:

```
myApp.nativeUI.createWindowMenu = function () {
  // Create window menu
  var windowMenu = new myApp.gui.Menu({ type: 'menubar' });
  if (process.platform === 'darwin'){
    // Create default Mac OS menus
    windowMenu.createMacBuiltin(myApp.name, {
      hideWindow: true
    });
  } else {
    // Create default file menu for Windows and Linux
    var fileMenu = new myApp.gui.MenuItem({
      label: "File"
    });
    var fileSubmenu = new myApp.gui.Menu();
    fileSubmenu.append(new myApp.gui.MenuItem({
      label: 'Exit',
      key: "x",
      modifiers: "ctrl",
      click: function () {
        myApp.gui.App.closeAllWindows();
      }
    }));
    fileMenu.submenu = fileSubmenu;
    windowMenu.append(fileMenu);
    // Create default edit menu for Windows and Linux
    var editMenu = new myApp.gui.MenuItem({
      label: "Edit"
    });
    var editSubmenu = myApp.nativeUI.createEditMenu();
    editMenu.submenu = editSubmenu;
    windowMenu.append(editMenu);
  }
}
```

```
    }  
    // Assign the menu to the window  
    myApp.mainWindow.menu = windowMenu;  
  };
```

First, we created the Window menu object, and then if the current platform is Mac OS X (Darwin refers to Mac OS), we call `createMacBuiltin()`; otherwise, we create a custom **File/Edit** menu for Microsoft Windows and Linux.

The code is pretty basic; however, you've probably noticed that in order to create the edit submenu, we're calling a custom `myApp.nativeUI.createEditMenu()` function. I kept it separated as we'll use the same menu for the Tasks context menu. Let's declare the function at the end of the `nativeui.js` file:

```
myApp.nativeUI.createEditMenu = function () {  
  var editMenu = new myApp.gui.Menu();  
  editMenu.append(new myApp.gui.MenuItem({  
    label: 'Cut',  
    click: function() {  
      document.execCommand("cut");  
    }  
  }));  
  editMenu.append(new myApp.gui.MenuItem({  
    label: 'Copy',  
    click: function() {  
      document.execCommand("copy");  
    }  
  }));  
  editMenu.append(new myApp.gui.MenuItem({  
    label: 'Paste',  
    click: function() {  
      document.execCommand("paste");  
    }  
  }));  
  editMenu.append(new myApp.gui.MenuItem({  
    label: 'Select All',  
    click: function() {  
      document.execCommand("selectAll");  
    }  
  }));  
  return editMenu;  
};
```

There's nothing fancy here; we're simply creating a default **Edit** menu, where each menu item corresponds to a document command.

Now, let's get back to the Window menu. We still have to create the **Options** menu. Right before the last line of code of `myApp.nativeUI.createWindowMenu()`, before assigning `windowMenu` to `myApp.mainWindow.menu`, add the following:

```
// Create options menu
var optionsMenu = new myApp.gui.MenuItem({
  label: "Options"
});
// Options submenu
var options = new myApp.gui.Menu();
```

We've created the submenu, now we need to add the individual submenu items. Let's see it step by step:

- Options menu item:

```
options.append(new myApp.gui.MenuItem({
  label: 'Options',
  key: "o",
  modifiers: "ctrl-alt",
  click: myApp.nativeUI.openOptionsWindow()
}));
```

When clicking on the Options menu item, we call `myApp.nativeUI.openOptionsWindow()`. We will see that in a minute, but, for the moment, let's see the other menu voices.

- Hide completed menu item:

```
options.append(new myApp.gui.MenuItem({
  label: 'Hide completed',
  type: 'checkbox',
  key: "h",
  modifiers: "ctrl-alt",
  checked: (localStorage.hideCompleted === 'true'),
  click: function () {
    localStorage.hideCompleted = this.checked;
    myApp.todos.load();
  }
}));
```


This menu item acts as a checkbox, with an on/off status and leverages `localStorage` to store its status. That's why the checked property is set based on the `localStorage.hideCompleted` value. So, when we click on the item, two things happen: first, `localStorage.hideCompleted` is set to the current menu status and then `myApp.todos.load()` is called. If you check it again, you'll see that `myApp.todos.load()` checks indeed for the `localStorage.hideCompleted` value to decide whether to load all the tasks or only the uncompleted one. As we stated in *Chapter 4, Data Persistence Solutions and Other Browser Web APIs*, *web storage is probably not the best data persistence solution, but it's probably the best one to store simple configuration data.*

- Export menu item:

```
options.append(new myApp.gui.MenuItem({
  label: 'Export',
  key: "e",
  modifiers: "ctrl-alt",
  click: function () {
    $('#fileDialog').click();
  }
}));
```

The export menu item click event produces a very simple effect. It simulates a click on the file input field inside `views/app.html` in order to open the save file dialog.

- Sync Menu item:

```
options.append(new myApp.gui.MenuItem({
  label: 'Sync',
  enabled: Boolean(localStorage.couchDbUrl),
  key: "s",
  modifiers: "ctrl-alt",
  click: myApp.todos.sync
}));
```

The sync menu item click event simply calls `myApp.todos.sync()`, which we have seen before in the application layer. An interesting thing to note is that the menu item gets disabled if `localStorage.couchDbUrl` is empty when the item is created.

Once we've created all the submenu items, we can finally assign the menu to the window menu with the following code:

```
optionsMenu.submenu = options;
windowMenu.append(optionsMenu);
```

This latest piece of code assigns the submenu to optionsMenu and then adds optionsMenu to the Window menu.

Implementing the Context menu

What we want to implement here is an **Edit** context menu to be shown when right-clicking on the content of our tasks. In order to implement this, we can leverage `jQuery.on` by passing the coordinates of the click to `contextMenu.popup()`. Let's see the actual implementation:

```
myApp.nativeUI.addContextMenuToTodos = function () {
  $('body').on("contextmenu", '.todo-content', function(e) {
    e.preventDefault();
    var contextMenu = myApp.nativeUI.createEditMenu();
    contextMenu.popup(e.originalEvent.x, e.originalEvent.y);
  });
};
```

We've already taken care of implementing `myApp.nativeUI.createEditMenu()`, so we can add the preceding function after `myApp.nativeUI.createWindowMenu()` and add a call to it into `myApp.nativeUI.init()`.

Restoring the window position

We haven't saved the window position yet, but we can already implement the code to restore it. In order to do so, we can leverage `localStorage`:

```
myApp.nativeUI.restoreWindowPosition = function () {
  if (localStorage.window_x !== 'undefined'){
    myApp.mainWindow.x = localStorage.window_x;
    myApp.mainWindow.y = localStorage.window_y;
    myApp.mainWindow.width = localStorage.window_w;
    myApp.mainWindow.height = localStorage.window_h;
  }
};
```

The code above is pretty simple; I'm simply checking whether `localStorage.window_x` has been set before. Then, if yes, I set the Main window coordinates and size. As before, remember to add a call to `myApp.nativeUI.restoreWindowPosition()` into `myApp.init()`.

Implementing the Options window

While creating the menu, we added an Options menu item that, once clicked, calls `myApp.nativeUI.openOptionsWindow()`. Let's implement it:

```
myApp.nativeUI.openOptionsWindow = function () {
  if (myApp.optionsWindow)
    return false;
  myApp.optionsWindow = myApp.gui.Window.open('options.html', {
    position: 'center',
    width: 400,
    height: 220,
    resizable: false,
    focus: true,
    toolbar: false
  });
  myApp.mainWindow.on('focus', function () {
    myApp.optionsWindow.focus();
  });
  myApp.optionsWindow.on('closed', function () {
    myApp.optionsWindow = null;
    myApp.mainWindow.removeAllListeners('focus');
  });
};
```

First, we verify that the Options window is not already opened and then we call `myApp.gui.Window.open()` by passing the name of the file to open (`options.html`) and giving it some options, including the focus flag.

Once the Options window has been opened, we listen for the focus event on the Main window and call the `focus` method in the Options window. This is a kind of hack as we're preventing the Main window from getting focus when the Options window is opened.

In the last rows, we follow the golden rule of assigning `null` to disposed objects, so when the Options window is closed, we assign `null` to it and then unregister the previously declared `focus` event bound on the Main window.

Now, we need to implement the actual Options window, so let's open `views/options.html` and define the HTML template:

```
<form>
  <label>
    Notes Background color
    <input type="color" id="notesColor" value="{{notesColor}}"
      list="color"/>
    <datalist id="color">
      <option value="#F8F8FF"></option>
      <option value="#d7e5c5"></option>
      <option value="#c8c3e0"></option>
      <option value="#cbdde1"></option>
    </datalist>
  </label>
  <label>
    Remote CouchDB Url
    <input type="text" id="couchDbUrl" value="{{couchDbUrl}}"
      placeholder="https://xxx.iriscouch.com/todo"/>
  </label>
  <div class="buttons">
    <button id="dismiss">Dismiss</button>
    <button id="save">Save</button>
  </div>
</form>
```

As you probably expected, it's a very simple form with two fields on it. The **Notes Background color** input field is a color picker; you can see we've associated it with the `color` `datalist` element in order to suggest a few predetermined colors to the user. The **Remote CouchDB Url** input field will store the URL of the remote CouchDB database.

Eventually, there are two buttons—one to save the data and the other to close the Window without saving.

Now, you can open `js/options.js` and start writing some code:

```
var myApp = {options:{}};
myApp.options.init = function () {
  var gui = require('nw.gui');
  myApp.optionsWindow = gui.Window.get();
  myApp.loadTemplate('options', {
    notesColor: localStorage.notesColor || '#F8F8FF',
    couchDbUrl: localStorage.couchDbUrl
  }, '#mainView');
```

```
myApp.options.listenForEvents();
};
myApp.options.listenForEvents = function () {};
$(window).load(myApp.options.init);
```

On the jQuery window load, we call the `init` function, which requires Native UI APIs, stores the current Window object, and then load the `views/options.html` template into the view.

Once the template has been loaded, we can start listening for events, so let's add the following inside `myApp.options.listenForEvents()`:

```
var softClose = false;
myApp.optionsWindow.on('close', function () {
    if (!softClose && confirm("Save the data before closing?"))
        myApp.saveData();
    this.hide();
    this.close(true);
});
$('#save').click(function() {
    myApp.saveData();
    softClose = true;
    myApp.optionsWindow.close();
});
$('#dismiss').click(function() {
    softClose = true;
    myApp.optionsWindow.close();
});
```

In the first line, we declare a variable named `softClose`. It will be used to check whether the closing action has been triggered by a button, by quitting the window with **x**, or by using keyboard shortcuts. As illustrated, the first thing we check on the `close` event is its value. If `softClose` is set to `false` (hence you are closing the window the hard way), a confirm dialog will ask the user if they are willing to save the data before closing the window. As we have seen in *Chapter 2, NW.js Native UI APIs*, the Options window close event will also be triggered by `App.closeAllWindows()`. So, this behavior will also apply when we try to close the Main window when the Options window is still open.

After checking for the `close` event, we can finally trigger an action when clicking on the **Save** or **Dismiss** button. We haven't seen yet what `myApp.options.saveData()` does, but it's actually pretty simple:

```
myApp.options.saveData = function () {
    // Using window to avoid context issue when closing
```

```
    window.localStorage.notesColor = $('#notesColor').val();
    window.localStorage.couchDbUrl = $('#couchDbUrl').val();
};
```

As illustrated, we're using `window.localStorage` instead of simply `localStorage`. That's to avoid issues when closing secondary windows as there appears to be some issues accessing the window context in the process of closing.

Now our Options window is up and working, but if you try it now, you will notice that you have to close and reopen the application for the changes to be applied.

We need a way to share the data between the Options and Main windows. There are clearly many ways to do so, but the easiest is probably to leverage the storage event.

As seen in *Chapter 4, Data Persistence Solutions and Other Browser Web APIs*, the storage event is triggered on concurrent open windows when a value is stored in `localStorage`, so we can use this event inside `js/nativeui.js` to listen for the changes applied by `myApp.options.saveData()`. Let's then add a call to `myApp.nativeUI.listenForStorageEvents()` into `myApp.nativeUI.init()` and then declare the function right before the end of the file:

```
myApp.nativeUI.listenForStorageEvents = function () {
    // Fired only when the event comes from a different window
    window.addEventListener('storage', function (e) {
        switch(e.key) {
            case 'notesColor':
                myApp.todos.changeBackgroundColor(e.newValue);
                break;
            case 'couchDbUrl':
                myApp.mainWindow
                    .menu.items[2]
                    .submenu.items[3]
                    .enabled = Boolean(e.newValue);
                break;
        }
    });
};
```

In the preceding function, we listen for the storage event. When the key on which we changed the value is `notescolor`, we run `myApp.todos.changeBackgroundColor()` by passing the new background color, whereas when it's `couchDbUrl`, we enable/disable the Sync menu item. The approach I used to change the property of the menu item is probably not the ideal one, as, if I change the order or position of the menu, it will break; however, I felt like I should give you this notion too. The alternative was obviously to reference the menu item somewhere on creation and then change its property at runtime.

Closing the application

As we've seen before, we can delay the closing of an NW.js application in order to save data or do an operation before actually quitting. In our case, there are three operations we need to implement when closing the NW.js application:

1. Give a chance to the Options window to save data
2. Store the window position
3. Compact the database

The first two operations are pretty clear, but you might wonder why we need to compact the database. Well, as amazing as it is to have a database that automatically syncs with a remote one, there's at least one downside. PouchDB stores a lot of extra data, which is used to keep the synchronization consistent. Much of this data, however, is not essential, so we can get rid of it when closing the application.

Add a call to `myApp.nativeUI.delayClosing()` into `myApp.nativeUI.init()` and then declare the function at the end of `nativeui.js`:

```
myApp.nativeUI.delayClosing = function () {
  myApp.mainWindow.on('close', function () {
    // Store window size and position
    if ( localStorage ){
      localStorage.window_x = myApp.mainWindow.x;
      localStorage.window_y = myApp.mainWindow.y;
      localStorage.window_w = myApp.mainWindow.width;
      localStorage.window_h = myApp.mainWindow.height;
    }
    // Hide window
    myApp.mainWindow.hide();
    // Close other opened windows
    myApp.gui.App.closeAllWindows();
    // Compact Db
  })
}
```

```
myApp.db.compact({}, function () {  
    // Actually close the Application  
    myApp.mainWindow.close(true);  
});  
});  
};
```

So, the first thing we do when the Main window is closing is store the window position. Before doing this, we check whether the `localStorage` object is set because, if for any reason, the application crashes, the `localStorage` object won't be available, and trying to access the property of an undefined object will prevent the application from closing.

At this point, we can hide the Main window and send a closing signal to the Options window with `App.closeAllWindows()`. Eventually, we compact our DB and when the compact operation is completed, we close the application.

Making the application open smoothly

We're almost done. The application should be working fine by now, but there's still a problem. When you open the application, you might notice a flickering problem. That depends on the fact that, at the moment, the application window is shown before our code is evaluated. As we've seen in *Chapter 2, NW.js Native UI APIs*, we can easily prevent that. Open the manifest file, set the `window.show` field to `false`, and then add the following line of code right at the end of `myApp.nativeui.init()`:

```
myApp.mainWindow.show();
```

This way, the application won't be shown until most of our code has been evaluated.

Summary

It has been quite a long journey to here, but we did it! You've built your first fully working NW.js application. Well, it's not perfect; there are many features that could still be implemented, and that's why I think it might be a good idea to keep adding a few more features, such as syncing with CouchDB on application start, close, and at regular intervals (measured in minutes); adding an expiration date to the single tasks; enabling the user to use the Markdown syntax inside the tasks content; and importing new tasks with JSON.

These are clearly just a few ideas, but they're actually pretty simple to implement in the NW.js environment.

Well, now our application is ready, but we still need to rely on NW.js to run it. We clearly need a better way to deploy it to our users. In the next chapters, we're going to see *how to package and deploy NW.js applications on different platforms* manually and even thanks to third-party packaging tools.

6

Packaging Your Application for Distribution

At this point, you should be clear about how the various parties involved in the implementation of an NW.js application fit together. Then, we can finally proceed with the packaging for the various platforms.

The packaging process can be simplified through the use of *automated packaging tools*. However, considering that NW.js is constantly evolving and that we cannot always be sure that the tool we have chosen for packaging is up to date with the version of NW.js in use, I believe it is appropriate to devote an entire chapter to the *manual process of packaging*.

In this chapter, we will look at the entire flow of packaging, step by step, which can be summarized in the following points:

- **Customizing the manifest file**
- **The general logic** behind packaging for multiple platforms
- Creating and customizing platform-specific configuration files
- Associating an **icon** with the application
- **Protecting the code**
- A brief digression on the **licensing** issue

Unfortunately, not all of the features that I'm going to introduce will be applicable on all the platforms. That's why, this chapter will be divided into platform-specific sections.

The manifest file

As we have seen in previous chapters, the manifest file is essential for the functioning of the application. Inside `package.json` are stored the general indications on how the application should be handled by NW.js. Let's start by taking a look at a minimal manifest file:

```
{
  "main": "index.html",
  "name": "My Application"
}
```

The preceding code is probably the minimum configuration required by NW.js, but there are many customizations possible. We can see all the possibilities listed here:

- **main:** This denotes the path of the main file to open when the project gets evaluated by NW.js.
- **name:** This indicates the unique name of the application. It's important that it is unique as it is also used to determine the name of the data path, which is the folder that will contain all the data of the application (for example, database, cache, and so on). It cannot contain spaces but only alphanumeric characters in addition to the `_`, `-`, and `.` characters.
- **version:** This is the application version (for example, 1.0.0).
- **nodejs:** This takes a Boolean value. It might become `iojs` in the upcoming versions of NW.js. By default, it is set to `true`, but when it's not, it will disable the support for Node.js to the application.
- **node-main:** As described in *Chapter 3, Leveraging the Power of Node.js*, this is the path to a Node.js JavaScript file that will be loaded and evaluated in the background.
- **single-instance:** This takes a Boolean value. By default, it is set to `true`, so only one instance of the same application can be run at the same time (for example, clicking on the application icon twice will not result in two windows being opened; only one window will open in response to the first click). If set to `false`, multiple contemporary instances will be allowed.
- **window:** This is an object containing a set of properties that allows us to customize the aspect and behavior of the main window. Most of these options can also be used with `Window` objects created through **Native UI APIs**:
 - **title:** This is the title of the window shown on the titlebar. It can be overwritten by the `<title/>` tag in the head of the HTML page.
 - **width:** This takes an integer value and indicates the starting width of the window.

- `height`: This takes an integer value and indicates the starting height of the window.
- `toolbar`: This takes a Boolean value. It determines whether or not the Chromium toolbar is supposed to be shown (including the **Dev Tool** button).
- `icon`: This is the path to the icon shown beside the application title in the titlebar and *also in the taskbar on Microsoft Windows*. (This is not shown at all on Mac OS X.)
- `position`: This is the starting position of the window on the screen. It can take one of three values: `null`, `center`, or `mouse`. The last two values correspond relatively to the center of the window or the current mouse position.
- `min_width`: This takes an integer value and indicates the minimum width that the window can be resized to by the user.
- `min_height`: This takes an integer value and indicates the minimum height that the window can be resized to by the user.
- `max_width`: This takes an integer value and indicates the maximum width that the window can be resized to by the user.
- `max_height`: This takes an integer value and indicates the maximum height that the window can be resized to by the user.
- `as_desktop`: This takes a Boolean value. On Linux systems with X11, this determines if the window should be set as the desktop background.
- `resizable`: This takes a Boolean value and indicates whether the window can be resized at all by the user.
- `always-on-top`: This takes a Boolean value and indicates whether the window should be shown on top of all the others in a permanent state.
- `visible-on-all-workspaces`: This takes a Boolean value. It permits us to make the windows accessible from all the workspaces only on Mac OS X and Linux.
- `fullscreen`: This takes a Boolean value and indicates whether the window should be opened in fullscreen (see *Chapter 2, NW.js Native UI APIs*).
- `show_in_taskbar`: This takes a Boolean value and indicates whether the icon of your application should be shown in the taskbar/dock (depending on the platform) or not.

- `frame`: This takes a Boolean value. When set to `false`, it removes the titlebar, the toolbar, and the resizing borders from the window.
- `show`: This takes a Boolean value and indicates whether the window should be shown on startup. As seen before, in most cases, it's best to set it to `false` than to show the window at runtime only when everything has been loaded (see *Chapter 2, NW.js Native UI APIs*).
- `kiosk`: This takes a Boolean value and indicates whether the window should be opened in the Kiosk mode (see *Chapter 2, NW.js Native UI APIs*).
- `transparent`: This takes a Boolean value and makes the window transparent in order to create custom-shaped screens.
- `webkit`: This is an object, the fields of which allow us to enable given components of WebKit; by default, all fields are set to `false`:
 - `plugin`: This takes a Boolean value and enables WebKit plugins such as Adobe Flash Player
 - `java`: This takes a Boolean value and enables support for Java Applets
 - `page-cache`: This takes a Boolean value and enables browser cache
- `user-agent`: This is a user agent string sent when HTTP calls are made. On the Wiki of the GitHub project, you can find a list of placeholders to attach specific information about the application (for example, `%name`, `%ver`, `%osinfo`, and so on).
- `node-remote`: This is a string containing a comma-separated list of hosts allowed to access Node.js functionalities (see the security section in *Chapter 4, Data Persistence Solutions and Other Browser Web APIs*).
- `chromium-args`: This is a string containing a list of space-separated parameters to be passed to Chromium in order to enable/disable given functions (for example, `--disable-accelerated-video` will disable the GPU acceleration).
- `js-flags`: This is a string containing a list of space-separated parameters to be passed to the V8 engine.
- `inject-js-start`: This is the path of a JavaScript file, the content of which are evaluated before the DOM gets built.
- `inject-js-end`: This is the path of a JavaScript file, the content of which is evaluated once the DOM is loaded before the `onload` event is fired.
- `additional_trust_anchors`: This is a string that contains a list of PEM-encoded certificates.

- `snapshot`: As we will see at the end of this chapter, it is possible to protect a part of our code. In this case, the `snapshot` field will be valued with the path of the compiled file.
- `dom_storage_quota`: This takes an integer value and indicates the number of megabytes assigned to the DOM storage.

Manifest file standards expect other fields to be assigned, which, however, are irrelevant for NW.js. Those fields are:

- `description`: This gives the description of the project
- `keywords`: This is an array of keywords describing the project
- `maintainers`: This is an array of maintainers of the project
- `contributors`: This is an array of contributors to the project
- `bugs`: This is the URL for submitting possible bugs
- `licenses`: This is an array of licenses adopted by the project
- `repositories`: This is an array of repositories from which to download the project
- `dependencies`: This is an object containing a list of dependencies that are installable through `npm install`

The general logic behind the packaging procedure

Generally, we can say that packaging an NW.js application is sufficient to:

- **Linux and Microsoft Windows**: Copy all the project files inside the folder containing the NW.js executable and deploy the entire folder
- **Mac OS X**: Copy all of the project files into `nwjs.app/Contents/Resources/app.nw/` and deploy the `nwjs.app` file

If this rule is generally valid, it is also true that it is just one of the many ways to package an NW.js application and does not provide any customization of the appearance and behavior of the application.

It should also be considered that some of the installed *Node.js/io.js modules might not work* or require different dependencies based on the current platform. For this reason, you should always redo `npm install` on any platform to which you plan to deploy.

Another thing has to be considered when naming files and folders. While Mac OS X and Microsoft Windows are case insensitive, this is not true for Linux. Moreover, when packaging NW.js applications, you must always respect **case sensitiveness**; otherwise, the application will not work, even on the case-insensitive platforms that were previously mentioned.

Finally, here's an important note on the approach employed by this book to illustrate the packaging step. The applications that we are going to build are **standalone** (or portable), which means that they do not need an installer to run.

Unfortunately, if on some platforms, you can implement most of the OS-related features directly into the application executable, on others, this might be very difficult and beyond the scope of this book.

A simple example may be registering a given file type to our application on various platforms. On Mac OS X, that's pretty straightforward as all you need to do is to edit the `Info.plist` file provided with the package, while on Microsoft Windows, you'll need to edit the system registry with different approaches depending on whether the file extension has already been registered or if it's a new one.

Every time you find yourself in the situation previously described, you have many possible ways to go on; the simplest is probably to do the following:

1. Create an installer for your application using third-party tools. In our example, we could adopt Inno Setup (<http://www.jrsoftware.org/isinfo.php>) in order to edit the system registry during the installation procedure.
2. Implement the feature directly into the application code leveraging the power of Node.js. In our example, we could use the Windows Node.js module (<https://github.com/Benvie/node-Windows>) to register the file association on the first application run.

This is also true if you intend to register system services, link the application to the application launcher, or access any other feature not easily implementable at the application level.

Being able to properly deploy an application to the various platforms requires very good and specific knowledge of how each graphic environment works; *NW.js Essentials* will not go deeper into the topic more than necessary, but it seemed important to clarify the reasons. However, case by case, I'll try to give you some hint on the next steps to follow in order to make your application look and feel like a native one.

In the next part of the chapter, we assume that you have already downloaded NW.js for every platform, as described in *Chapter 1, Meet NW.js*. The following procedures are very specific to each operating system, so each procedure must be executed on the target platform.

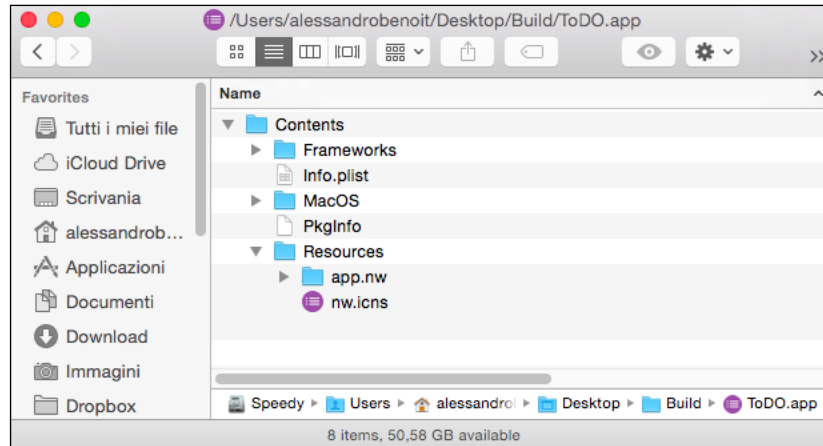
In order to make the procedures more understandable, we will pretend we are packaging our example TODO application.

Packaging NW.js applications for Mac OS X

In order to package an NW.js application for Mac OS X, proceed as follows:

1. Create a copy of `nwjs.app` and rename it to `TODO.app`.
2. Right-click on the `TODO.app` file and select **Show Package Content**.
3. A finder window should open; open the `Contents/Resources` folder.
4. Create a new folder, name it `app.nw`, and then copy all the content of your TODO project into it (including the manifest file).
5. If your project is large enough, you could consider to zip the content of the `app.nw` folder to an archive inside the `Contents/Resources` folder, delete the `app.nw` folder, and rename the archive to `app.nw`, but be aware that uncompressing its content at each run might slow down the application launch time.
6. Change the default icon located in `Contents/Resources/nw.icns` with one for your application. To create an `*.icns` icon, you can start with a transparent PNG icon of the dimensions 512 x 512 pixels or 1024 x 1024 pixels (retina) and then use **img2icns** (<http://www.img2icnsapp.com/>) or some online service such as <http://www.iconverticons.com/> to convert it to an `*.icns` file.

7. At this point, the `ToDo.app` content should look as it does in the following screenshot:



8. Open the `Contents/Info.plist` file inside `ToDo.app` with your favorite editor and then apply the following changes:
 1. Edit `CFBundleName` with the name of your application:

```
<key>CFBundleName</key>
<string>ToDo</string>
```
 2. Edit `CFBundleShortVersionString` and `CFBundleVersion` with the version of your application:

```
<key>CFBundleShortVersionString</key>
<string>1.0.1.0</string>
<key>CFBundleVersion</key>
<string>1.0</string>
```

Now, as a simple hack to bypass the system icon cache, you can move the `ToDo.app` file to a different position, and you should see the new icon applied. In order to be sure that everything is alright, start the application and verify that the application name is shown in the top-left corner of the window menu in place of the default **nwjs** menu:



Now, you can click on the **ToDo** menu, then on **About ToDo!**, and in the small window that just opened, you should see the correct application name and version, much like in the following screenshot:



Associating a file extension with your application

In order to associate a given file extension to your application, you can open the `Contents/Info.plist` file and add the following code to the `CFBundleDocumentTypes` section:

```
<dict>
  <key>CFBundleTypeIconFile</key>
  <string>nw.icns</string>
  <key>CFBundleTypeName</key>
  <string>ToDo</string>
  <key>CFBundleTypeRole</key>
  <string>Viewer</string>
  <key>LSHandlerRank</key>
  <string>Owner</string>
  <key>CFBundleTypeExtensions</key>
  <array>
    <string>todo</string>
  </array>
  <key>LSIsAppleDefaultForType</key>
  <true/>
  <key>CFBundleIconFile</key>
  <string>nw.icns</string>
</dict>
```

Now, if we create a new file with the `.todo` extension and if the extension has not already been used by another application, it will get the icon described in the `CFBundleIconFile` field, and it will automatically be opened by our application (see *Chapter 2, NW.js Native UI APIs*, on how to handle file opening with NW.js).

Packaging NW.js applications for Microsoft Windows



Before you start

If you are planning to add a shortcut to the application's screen, you should consider using `gui.App.createShortcut()` as described about the Web Notifications API at the end of *Chapter 4, Data Persistence Solutions and Other Browser Web APIs*.

Let's see all the steps necessary to package an NW.js application for Microsoft Windows:

1. Create a copy of the downloaded NW.js folder and rename it to `ToDo`.
2. Delete unnecessary files such as `nwjc.exe` and, only if you are not using media libraries, `ffmpegsumo.dll`.
3. Zip the content of the `ToDo` project folder, the one containing the `package.json` file, in an `app.zip` file (only the content of the folder; if you'd try to zip the entire folder, it won't work).
4. Rename the zip file to `app.nw` (if the file extension is hidden, navigate to **View | Folder Options**, and remove the flag from the **Hide extensions for known file type** option).
5. Copy the `app.nw` file into the NW.js folder that was renamed to `ToDo`.
6. Open the command prompt inside the folder and type the following commands:

```
C:\...\ToDo> copy /b nw.exe+app.nw todo.exe
```

```
C:\...\ToDo> del nw.exe
```

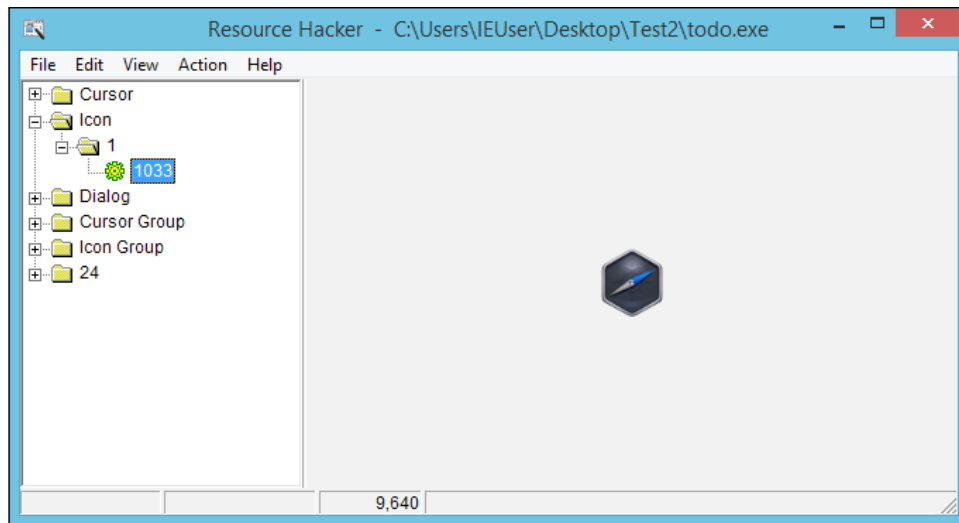
```
C:\...\ToDo> del app.nw
```

About renaming the NW.js executable



The previous command will merge the NW.js executable file with `app.nw` into a new `todo.exe` file and delete the older files. If, for any reason, you are using native Node.js/io.js modules, the resulting executable would not be renamed into `whatever.exe`, but will need to stay as `nw.exe`.

7. Download, install, and open **ResourceHacker** at <http://www.angusj.com/resourcehacker/>.
8. Open `todo.exe` with ResourceHacker and select the resource corresponding to the application icon. Check the following screenshot for reference:



9. Right-click on the name of the resource (in our example, 1033) and select **Replace Resource**.
10. On the window that has just opened, click on **Open file with new Icon**, select your `*.ico` file (you can create one online at <http://www.iconverticons.com/> starting from a 256 x 256 pixel transparent PNG file), and click on **Replace**.
11. Eventually, you can click on **File** and then **Save**. As a side effect, a copy of the original executable file will be created with the `_original` suffix. Verify that the application works properly before deleting it.

If the icon has not changed already, it probably depends on the Microsoft Windows icon cache. To clean it up, you can simply type the following commands in the command prompt:

```
C:\>ie4uinit.exe -ClearIconCache  
C:\>taskkill /IM explorer.exe /F  
C:\>explorer
```



Remember that the icon shown in the Microsoft Window taskbar is not the one just being applied but the one specified in the `window.icon` field inside the manifest file.

Registering a file type association on Microsoft Windows

In order to register a file type association on Microsoft Windows, check the *The general logic behind the packaging procedure* section. The registry keys you'll have to add/edit are under `HKEY_CLASSES_ROOT`. It took me a bit of time to understand how Microsoft Windows handles this type of associations, and the best hint I can give you is to check how other applications handle file/mime type association.

Packaging NW.js applications for Linux

On Linux, things got even trickier as I figured out that the concept of standalone applications is not very popular on Linux platforms.

The following procedure will enable you to build an executable for your application; however, the steps described to add an icon will not be replicable unless you build a Linux installable package in one of the many possible formats: `*.pkg`, `*.rpm`, `*.deb`, `*.tgz`, and so on.

Unfortunately, it is far beyond the purpose of the book to explain how such packaging systems work, but let's deal with one problem at a time:

1. Create a copy of the downloaded NW.js folder and rename it to `ToDo`.
2. Delete `nwjc` and `libffmpegsumo.so` if you don't need media libraries.
3. From within the `ToDo` project folder, execute the following command to create a zip file named `app.nw`:

```
$ zip -r ../app.nw *
```

4. The `app.nw` file will be placed just outside the project folder; copy it inside the `ToDo` folder.
5. Execute the following commands from within the `ToDo` folder in order to merge the `nw` executable with your project archive:

```
$ cat nw app.nw > todo
$ chmod +x todo
$ rm nw
$ rm app.nw
```
6. You are done! If you double-click on the `todo` file, the application will open as expected.

Adding icon and file type associations on Linux

As anticipated, now we have to deal with the association of an icon with our `todo` executable. The procedure described as follows will introduce you to the creation of a `*.desktop` file (desktop files are sometimes also referred to as **application launchers**), the standard used to specify the behavior of programs running on **X11**:

1. Create a `todo.desktop` file inside the `/usr/share/applications/` folder and open it with your favorite code editor.
2. Paste the following content into the file:

```
[Desktop Entry]
Version=1.6
Name=ToDo
Comment=ToDo Application
Exec=/path/to/todo $U
Icon=/path/to/icon.png
Terminal=false
Type=Application
Categories=Utility;Application;
```
3. Once you've saved the file, you need to make it executable with the following command:

```
$ sudo chmod +x todo.desktop
```
4. Eventually, you'll find it in the launcher with the specified icon.

The problem with this approach, as you probably guessed already, is that you cannot know the path to the application until you're actually on the user's system, so you'll need some kind of installer/packager to make it work.

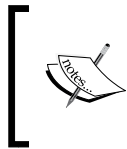
This goes for file type associations as well. In order to make your application handle a given file type, you must edit (or create) one of the following files depending on the targeted Linux distribution:

- `~/.local/share/applications/mimeapps.list`
- `~/.local/share/applications/defaults.list`

Add the following instructions in order to associate plain text files with your application:

```
[Default Applications]
text/plain=todo.desktop
```

You can find more information on X11 `.desktop` files at https://wiki.archlinux.org/index.php/Desktop_entries.



I hope this section wasn't too messy. I've a pretty good knowledge of Mac OS X and Linux CLI, but I feel kind of powerless when dealing with Microsoft Windows or X11. Hopefully, you'll know the subject much better than I do so you'll be able to put the missing pieces together.

Securing your source code

As I explained in the introduction, NW.js provides a way to protect part of your source code using `nwjc` (formerly `nwsnapshot`), which you should be able to find within the downloaded NW.js archive. This is still an *experimental feature*, so you should take extra care when using it.

Your code must be compiled with `nwjc` during the packaging process; then it's processed at runtime when the application gets executed. This will have a remarkable operational cost, and it will *slow down the performances of your code by around 30 percent* (based on V8 Benchmark suite results, as reported on the Wiki of the project).

Moreover, *the compiled code will not be cross-platform nor compatible between different versions of NW.js*, so you'll have to recompile your code at every update of NW.js for each operating system. Another limitation of `nwjc` is that there is a limit on the size of the source code to be compiled, which is around 250 KB.

Once we've established all the limitations of `nwjc`, let's dive deep into its functioning. In order to compile some code, you will have to execute the following command from your terminal / command prompt inside the project folder on Mac OS X or Linux:

```
$ path/to/nwjc --extra_code source.js snapshot.bin
```

Execute the following command on Microsoft Windows:

```
C:\...\project> C:\path\to\nwjc.exe --extra_code source.js snapshot.bin
```

This command will create a new `snapshot.bin` file within your project folder. Then, you can make it load through your NW.js application by editing the manifest file as follows:

```
{
  "name": "nw-demo",
  "main": "index.html",
  "snapshot" : "snapshot.bin"
}
```

It's very important you remember that the code compiled into `snapshot.bin` will be evaluated before the application launches, so neither the `window` object nor the `document` object will be instantiated at that time. In order to avoid headache, it would be better to only declare functions and variables in `snapshot.bin` rather than run your code.

However, for the sake of learning, all of the following code would be successfully evaluated:

- **source.js » snapshot.bin:**

```
var variable = 'Hello';
(function () {
  variable += ' World';
})();
function writeHelloWorld () {
  console.log(variable);
  localStorage.string = variable;
  document.getElementById('main').innerHTML =
    localStorage.string;
}
```
- **index.html:**

```
<div id="main"></div>
<script type="text/javascript">
  writeHelloWorld();
  console.log('Accessing variable', variable);
</script>
```

Inside `snapshot.bin`, you can also require Node.js modules and work with Native UI APIs, always remembering that most of the objects are not available when this part of the source code is being evaluated.

About NW.js application licensing

I'm no lawyer and you shouldn't trust me so much on this; however, I felt I should give you at least some hints on how to deal with NW.js application licensing.

First, I quote what is stated on the GitHub Wiki page of the project:

"Since the binary is based on Chromium, multiple open source license notices are needed including the MIT License, the LGPL, the BSD, the Ms-PL and an MPL/GPL/LGPL tri-license."

You can find most of these at <http://www.gnu.org/licenses/license-list.en.html>.

From my understanding, the MPL/GPL/LGPL tri-license is a license that lets you choose one of the three as a better fit for your project, so **you don't need to release your code under an open source license** as established by the GPL license, but you can go with a commercial license.

I quote the online documentation of the NW.js project again:

"This doesn't apply to your code and you don't have to open source your code"

Summary

With the packaging process, we're almost at the end of our journey. By now, you should feel comfortable customizing the manifest file and playing with all the options dedicated to the window's appearance.

You may have already packaged your application for the many possible operating systems, probably struggling here and there trying to implement platform-specific features, but well (you're going to hate me for saying it), that's all part of the learning process.

Most of all, you have not only discovered how easy is to build a full, working executable for each platform, but also how time consuming it can be, so we will devote the next chapter to a proper understanding of some of the most promising automated packaging tools.

7

Automated Packaging Tools

As illustrated in the previous chapter, even though it is pretty easy to build an NW.js package for a given platform, it might become easily frustrating to repeat the procedure for each platform every time you plan to release an update. That's why over time, many good and willing developers have built **automated packaging tools** using different technologies on different platforms.

In this chapter, we're going to analyze the pros and cons of some of the most promising ones and describe all the steps required to build full, working NW.js packages in no time. Some of these tools come with a GUI, while others are standalone CLI or Grunt plugins. It is up to you to choose the one among these that better fit your development workflow.

Web2Executable

The download link for **Web2Executable** is <https://github.com/jyapayne/Web2Executable>.

Web2Executable is a pretty smart NW.js packaging tool written in Python by Joey Paine, a Canadian tech guy. Between the many tools I will introduce you, this is the only one that comes with a quite essential but really usable GUI (if you are not interested in the GUI, you can also download the CLI version).

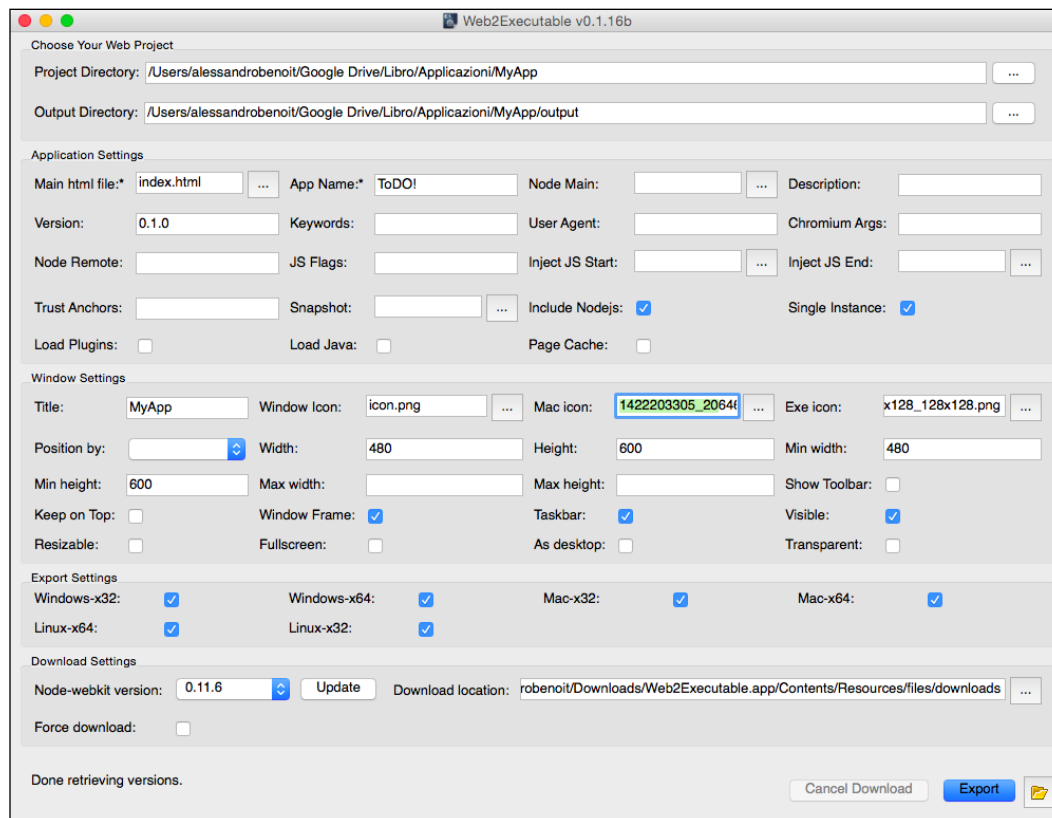
Web2Executable is available to all the three platforms: Mac OS X, Microsoft Windows 7 +, and Linux (tested on Ubuntu 14.04).

If you decide to run it out of the source code, you will need Python, PySide, and PIL; however, if you download one of the prepackaged executables, it will work out of the box.

What Web2Executable does can be summarized as follows:

- Downloading the NW.js package for the given version of the selected platform
- Building packages for various platforms
- Eventually adding the icon to the Mac OS X and Microsoft Windows executable

Let's take a look at what Web2Executable looks like:



Let's go through Web2Executable's step-by-step building procedure:

1. In the top panel, choose the project folder path and the destination directory (by default, an output directory within the project folder).
2. If you haven't done it already, set the manifest file options in the **Application Settings** and **Window Settings** panels. You would probably uncheck the **Show Toolbar** option, which I always forget to hide when building.

3. If you are planning to deploy for Mac OS X or Microsoft Windows, in the **Window Settings** panel, set the Mac or Exe icon.
4. In **Export Settings**, choose the platforms you want to deploy for; the relative NW.js versions will be downloaded consequently. Web2Executable makes it possible to build packages for multiple platforms at a time, but remember that, this way, Node.js/io.js modules will not be rebuilt and compatibility issues could arise.
5. In the **Download Setting** panel, choose the version of NW.js in use and where to download all the packages. (You'll need to know this path in case you need to customize NW.js packages.)
6. Eventually, click on the **Export** button to make Web2Executable do its magic.

All the options set in the preceding steps will be stored in your manifest file (`package.json`). Those that are specific to Web2Executable will be stored in a `webexe_settings` object, while a few custom ones will be added to the `window` object for the Mac or Exe icon.

At the moment, Web2Executable will not take care of customizing the `Info.plist` file within the Mac OS X package, so you'll have to do it manually (as described in *Chapter 6, Packaging Your Application for Distribution*); however, the developer assured me that he's already on it.



Error while loading shared libraries: libudev.so.0

When building for Linux, you might encounter the `libudev.so.0` error. In that case, you'll have to unzip the NW.js Linux package downloaded by the tool, run the procedure described in *Chapter 1, Meet NW.js*, and eventually zip the package back. This goes for all the other tools reviewed in this chapter.

node-webkit-builder and grunt-node-webkit-builder

The project links for `node-webkit-builder` and `grunt-node-webkit-builder`, respectively, are as follows:

- <https://github.com/mllrsohn/node-webkit-builder>
- <https://github.com/mllrsohn/grunt-node-webkit-builder>

node-webkit-builder is a Node.js script that can be used both as a module and a CLI tool.

As you guessed already, it also comes in a **Grunt plugin version**, while if you are a **Gulp** user, you can just make use of the module.

node-webkit-builder is one of the most complete tools out there as it takes care of packaging, adding icons, and filling the `Info.plist` data for Mac OS X, fast and offers a lot of possible customizations.

Unfortunately, many of these great customization options are not accessible through the CLI at the moment. Waiting for those features to be integrated, the CLI can still come in pretty useful for testing, so let's take a look at how it works.

First of all, install the module globally using:

```
$ npm install node-webkit-builder -g
```

Once the module has been installed, you can run the `nwbuild` command as follows:

```
$ nwbuild [options] [path]
```

Whereas `path` is the path to your project folder, `options` are the ones described in the following code:

```
-p Operating System to build ['osx32', 'osx64', 'win32', 'win64']
-v NW.js version [default: "latest"]
-r Runs NW.js project [default: false]
-o The path of the output folder [default: "./build"]
-f Force download of node-webkit [default: false]
--quiet Disables logging
```

Let's see a couple of examples:

- **Run a project** (on the current platform):

```
$ nwbuild -r /path/to/the/project
```
- **Build a project for all platforms** (without icon support):

```
$ nwbuild -p win32,win64,osx64,linux32,linux64 /path/to/the/project
```

Running the previous command will lead node-webkit-builder to download all the needed NW.js packages, cache them, and eventually run or build the project. Running the CLI takes us to the following screen:

```

Downloading: http://dl.nwjs.io/v0.12.0-alpha3/nwjs-v0.12.0-alpha3-linux-x64.tar.
gz
  downloading [=====] 100% 0.0s

Create release folder in /Users/alessandrobenoit/Desktop/Built/ToDo!/osx32
Create release folder in /Users/alessandrobenoit/Desktop/Built/ToDo!/win64
Create release folder in /Users/alessandrobenoit/Desktop/Built/ToDo!/win32
Create release folder in /Users/alessandrobenoit/Desktop/Built/ToDo!/osx64
Create release folder in /Users/alessandrobenoit/Desktop/Built/ToDo!/linux32
Create release folder in /Users/alessandrobenoit/Desktop/Built/ToDo!/linux64

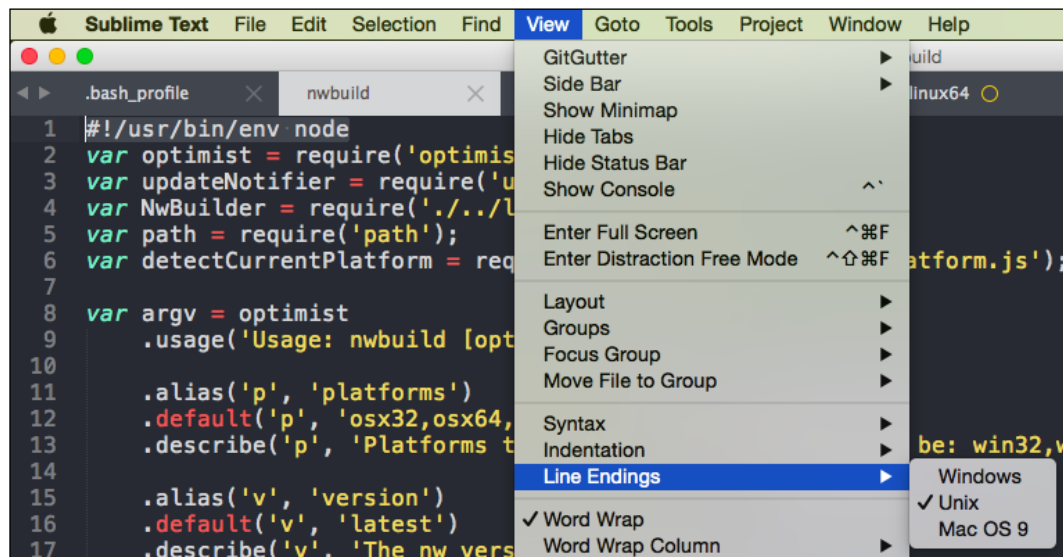
```

When running the CLI, you might get the following error:

```
env: node\r: No such file or directory
```

The issue comes from an error in how line endings are handled in the project source code. In order to fix it, you'll have to change the line endings on the node-webkit-builder/bin/nwbuild file.

On VIM, you can use the `:set ff=unix` command, while on Sublime Text, you can fix it by navigating to **View | Line Endings | Unix**:



As previously stated, at the moment, with the CLI, you cannot handle icons or any other customization. In order to do so, you'll have to create a Node.js script defining all the needed options.

Going further, I'm going to bring in a very simple example in order to make it more understandable. First, create a `myApp` folder and then populate it with the following files and subfolders:

```
myApp/  
  > project/  
    - nwbuild.js  
    - macCredits.html  
    - macIcons.icns  
    - winIcon.ico  
    - package.json
```

Move your project files to the project folder and then edit the `package.json` file as follows:

```
{  
  "name": "myAppBuilder",  
  "dependencies": {  
    "node-webkit-builder": "~1.0.8"  
  }  
}
```

Save the file and run `npm install` to install `node-webkit-builder` locally.

Now, edit the `nwbuild.js` file as follows:

```
var NwBuilder = require('node-webkit-builder');  
  
// Configuration  
var nw = new NwBuilder({  
  files: './project/**',  
  platforms: ['win', 'osx', 'linux'],  
  macIcns: './macIcon.icns',  
  macCredits: './macCredits.html',  
  winIco: './winIcon.ico'  
});  
  
// Handle logging  
nw.on('log', console.log);  
  
// Build returns a promise  
nw.build().then(function (a) {
```

```

    console.log('\033[32m' + '✓ All done!');
  }).catch(function (error) {
    console.error('\033[31m ✖ ' + error);
  });

```

In the preceding script, we load the module, provide a set of options, and eventually start the packaging operation and listen for its result.

In order to run it, type the following command in your prompt/terminal within the `myApp` folder:

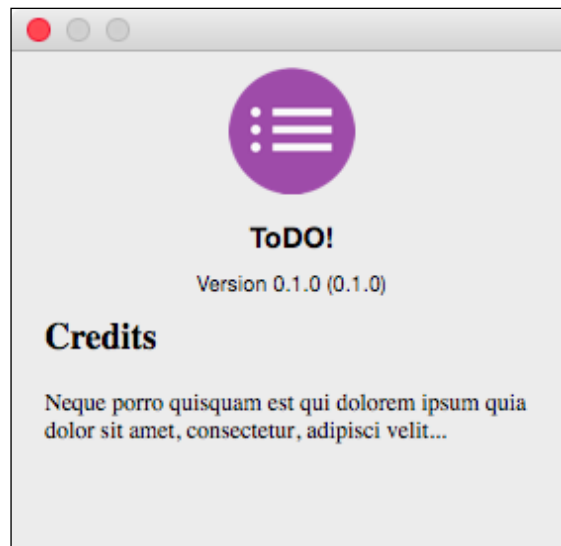
```
$ node nwbuild.js
```

Let's have a look at the whole set of available options to better understand how the preceding code works:

- `files`: This is the path to the project. As it supports *simple-glob*, you can declare multiple files, or even files and folders, that you don't want to be packaged (for example, `['foo/*.js', '!foo/bar.js', 'foo/bar.js']`).
- `version`: This is the version of NW.js you want to use. Each time you download a new version, it is cached in the `cache` folder within the root path of `myApp`. By default, the latest version will be used.
- `platforms`: This denotes the operating systems for which you want to build the project. You can use the specific platform (for example, `['osx32', 'osx64']`) or the whole group (for example, `['osx']`).
- `appName`: This is the name of the application used to name files and populate the Plist file. By default, the `name` property from the project manifest file will be used.
- `appVersion`: This is the version of the application used to name files and populate the Plist file. By default, the `version` property from the project manifest file will be used.
- `buildDir`: This is the path to the output directory. By default, a new `build` folder will be created within the root path of `myApp`.
- `cacheDir`: This is the path to the folder where the downloaded NW.js packages will be stored. By default, a new `cache` folder will be created within the root path of `myApp`.
- `buildType`: This is the convention used to name output packages. By default, it uses the plain `appName` option, but this option takes three other values:
 - **versioned**: `[appName] -v[appVersion]`
 - **timestamped**: `[appName] - [timestamp]`

- A function where options are passed as scope, for example:

```
function () {  
  return this.appVersion;  
}
```
- `forceDownload`: If this is set to `true`, it will clear all the cached NW.js packages and download everything back.
- `macCredits`: Only on Mac, you can set a custom HTML credit file, the content of which will be shown inside the **About Info Box** as follows:



- `macIcns`: This is the path to the icon that will be associated with the executable on Mac OS X.
- `macZip`: This indicates whether the `app.nw` folder should be zipped on Mac OS X. By default, it is set to `false`.
- `macPlist`: This is the path to the custom `Info.plist` file to replace the default one. You can even pass an object defining custom properties to be added to the default Plist file generated by node-webkit-builder.
- `winIco`: This is the path to the `*.ico` file to be associated with the executable for Microsoft Windows. In order for this option to work on Mac OS X and Linux, you'll have to install Wine from <http://winehq.org/> or run `brew install wine`.

One more thing to know about node-webkit-builder is that it lets you override the default manifest configuration with a custom set of options based on the destination platform. In order to do so, you will declare a `platformOverrides` object inside the project manifest file. Let's see a simple example:

```
{
  "name": "myApp",
  "version": "0.1.0",
  "main": "index.html",
  "window": {
    "height": 400
  },
  "platformOverrides": {
    "win": {
      "name": "myAppWin",
      "main": "indexWin.html"
    },
    "osx32": {
      "window": {
        "height": 500
      }
    }
  }
}
```

On the resulting packages, the `platformOverrides` option will be removed, but specific overrides will be applied to the main set of options. So, for example, both Win32 and Win64 manifest files will look something like the following:

```
{
  "name": "myAppWin",
  "version": "0.1.0",
  "main": "indexWin.html",
  "window": {
    "height": 400
  }
}
```

This is really convenient as it allows us to handle a new layer of customization to the building process, which would otherwise require multiple building scripts to target specific platform requirements.

grunt-node-webkit-builder

Now, let's take a look at how grunt-node-webkit-builder works:

1. Duplicate the previously created folder structure and edit `package.json` as follows:

```
{
  "name": "myAppBuilder",
  "dependencies": {
    "grunt": "^0.4.1",
    "grunt-node-webkit-builder": "~1.0.0"
  }
}
```

2. Save the file and run `npm install`; now you can create your `gruntfile.js`:

```
module.exports = function(grunt) {
  // Configuration
  grunt.initConfig({
    nodewebkit: {
      options: {
        platforms: ['osx', 'win', 'linux'],
        macIcns: './macIcon.icns',
        macCredits: './macCredits.html',
        winIco: './winIcon.ico'
      },
      src: ['./project/**'] // Your node-webkit app
    }
  });
  // Load plugin
  grunt.loadNpmTasks('grunt-node-webkit-builder');
  // Run
  grunt.registerTask('default', ['nodewebkit']);
};
```

On `grunt-node-webkit-builder`, you can use all the options seen for `node-webkit-builder`. The only difference is that you'll have to state the project folder in the `src` field.

3. Once you've saved the Grunt file, you can run it just using the following command:

```
$ grunt
```

**Mac OS X ulimit**

If you get an EMFILE error or a generic *too many open files* error, you'll have to run `ulimit -n 1024` from your terminal or put the command in your `~/.bash_profile` file.

As you can see, node-webkit-builder and grunt-node-webkit-builder are very complete and customizable solutions to build NW.js applications for multiple platforms at once, but, nevertheless, you should always remember the good practice of rebuilding Node.js/io.js modules on each platform.

generator-node-webkit

The project link for generator-node-webkit is <https://github.com/Dica-Developer/generator-node-webkit>.

generator-node-webkit is a **Yeoman generator** for NW.js. Unlike previously reviewed tools, it not only provides an easy way to package NW.js applications, but also *enforces good coding standards* and comes with *tools* and a *basic folder structure* to start your project from.

You're probably familiar with Yeoman already, but if you're not, it's a Node.js-based tool for scaffolding web applications, which is perfectly applicable to our case. You can easily install Yeoman globally with npm:

```
$ npm install -g yo
```

Once you're done with it, you can also install generator-node-webkit:

```
$ npm install -g generator-node-webkit
```

Eventually, create a new folder for your project and start the generator with the following command:

```
$ yo node-webkit
```

If everything went right, a screen, much like the one in the following screenshot, should appear:

```

  (o)
  ( 'U' )
  A
  ~
  |
  o
  Y

Welcome to Yeoman,
Ladies and gentlemen!

? What do you want to call your app? Allowed characters ^[a-zA-Z0-9]+$ myApp
? A little description for your app? Demo App
? Would you mind telling me your username on GitHub? micc83
? Do you want to install one of the node-webkit examples? No
  info Get GitHub informations
✓ Github informations successfully retrieved.
  invoke node-webkit:download
? Do you want to download node-webkit? Yes
? Please specify which version of node-webkit you want download: (v0.10.5)
  info Check if version "v0.10.5" is available for download.
? Please specify which version of node-webkit you want download: v0.10.5
? Which platform do you wanna support? MacOS 32, MacOS 64, Linux 64, Linux 32, Windows
  info Creating folder structure for node-webkit source.
✓ Created: "resources/node-webkit"
✓ Created: "resources/node-webkit/MacOS32"
✓ Created: "resources/node-webkit/MacOS64"
✓ Created: "resources/node-webkit/Linux64"
✓ Created: "resources/node-webkit/Linux32"
✓ Created: "resources/node-webkit/Windows"

```

As you can see, you'll be asked to answer a few questions; let's examine these step by step:

- **What do you want to call your app?**
It's the name of the application, used to valorize the manifest file, the resulting packages filenames, and the application name field in the Mac OS X Plist file.
- **A little description for your app?**
It's the app description used on many relevant places, such as in the resulting application packages.
- **Would you mind telling me your username on GitHub?**
It's the author's GitHub name used to add the copyright to the Mac OS X Plist file. (I assume that in the future, it will also be used elsewhere.)

- **Do you want to install one of the node-webkit examples?**

If the answer is yes, you'll be prompted to choose an example NW.js project to download.

- **Do you want to download node-webkit?**

If the answer is yes, you'll be prompted to choose the version and the platform. When you choose the version, remember to put a *v* before the number (for example, `v0.11.6`).

Once you've answered all the questions, your folder will be populated with many files and folders. Let's take a look at the ones that matter to the packaging workflow:

- `app/`: The project source code will go into this folder. A precompiled manifest file will be provided along with a few default folders: `css`, `views`, and `js`.
- `dist/`: This is the output folder for the packaging process.
- `resources/mac/`: In this folder, you will find all the assets needed to build the NW.js package for Mac OS X.
- `resources/node-webkit/`: In this folder, you will find all the unzipped NW.js packages (pretty useful when you need to fix the **libudev.so.0** error).
- `.editorconfig`: This is a configuration file for your IDE. It will set a few default rules to enforce good development practices. In order to make your IDE understand these rules, you'll have to download the appropriate plugin from <http://editorconfig.org/>.
- `.jshintrc`: This is a config file for **JSHint**. If you've never heard of this, you should definitely take a look at <http://jshint.com/>. (A plugin for Sublime Text is also available.)
- `bower.json`: Using Bower isn't mandatory; however, a boilerplate configuration file is provided.
- `Gruntfile.js`: Here's where all the magic happens. In order to build your application, you'll have to call the appropriate Grunt task, but we'll talk about that in a second.
- `package.json`: This is the manifest file for generator-node-webkit that is not to be confused with the one inside your project. The author information will be stored in this file.

Let's now take a look at the many available Grunt tasks:

```
$ grunt check
```

This line will perform a JSHint check of your code. This task will also be executed each time you call a `dist` task:

```
$ grunt dist-[platform]
```

This line will take care of the packaging process. The `platform` is the operating system you want to build the package for, and the available platforms are: Linux, Linux32, Win, Mac, and Mac32. It would serve no purpose to say that you can concatenate multiple `dist` tasks by adding a new task to the default Grunt file.

Unfortunately, at the moment, generator-node-webkit lacks the ability to associate the icon with the Microsoft Windows package. On the other hand, it has a few interesting options for Mac OS X when you run `$ grunt dist-mac`.

If an `nw.icns` icon file is found in the `resources/mac/` directory, it will be associated with the resulting package. Moreover, a custom `Info.plist` file will be created with the name, version, and copyright of your application, and you'll be able to customize it by editing `resources/mac/Info.plist`.

An interesting and unique feature of generator-node-webkit is that it implements the ability to build `*.dmg` files (**Apple disk image**) that are still one of the most common way to distribute Mac OS X applications out of the App Store.

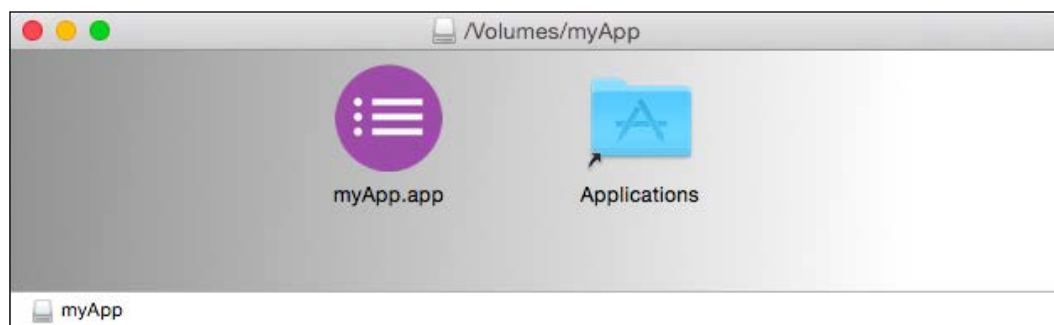
In order to build a `*.dmg` file for your application, you'll have to first build the Mac OS X version of the package and then run:

```
$ grunt dmg
```

A new `*.dmg` file will be created in `dist/MacOS64/` using the `background.png` file provided inside the `resources/mac/` folder for the background of the window.

Unfortunately, the `dmg` command will only work if it is executed on a Mac OS X platform.

Here's a screenshot of how the `.dmg` archive will look once it is open:





The first time I ran `grunt dmg` on version 1.0.3 of `generator-node-webkit`, I got the following error:

```
hdiutil: create failed - No such file or directory
```

If it isn't addressed by the publishing of the book, you can easily fix it by opening `resources/mac/package.sh` and renaming all the occurrences of the macOS pathname to the correct one (in my case, `MacOS64`).

Summary

In this chapter, I gave a quick overview of some of the most interesting packaging tools, but there are many others, such as **nodebob** and **lein-node-webkit-builder**.

I made one myself (**Nuwk!**) that has a few thousand downloads, but currently, I have no time to follow a project that anyway is in a very early stage of development.

I guess it would be pretty cool if a frontend guy (or girl) took the time to build a decent GUI around `node-webkit-builder`. If you think about it, most of you should be able to do it by now. It would be an additional incentive for newbies to approach `NW.js`.

That said, we can move on to the next subject: **debugging NW.js applications**. There are many approaches to debugging, and I cannot go deep into each one, so I will simply give you an overview of the tools needed for the job.

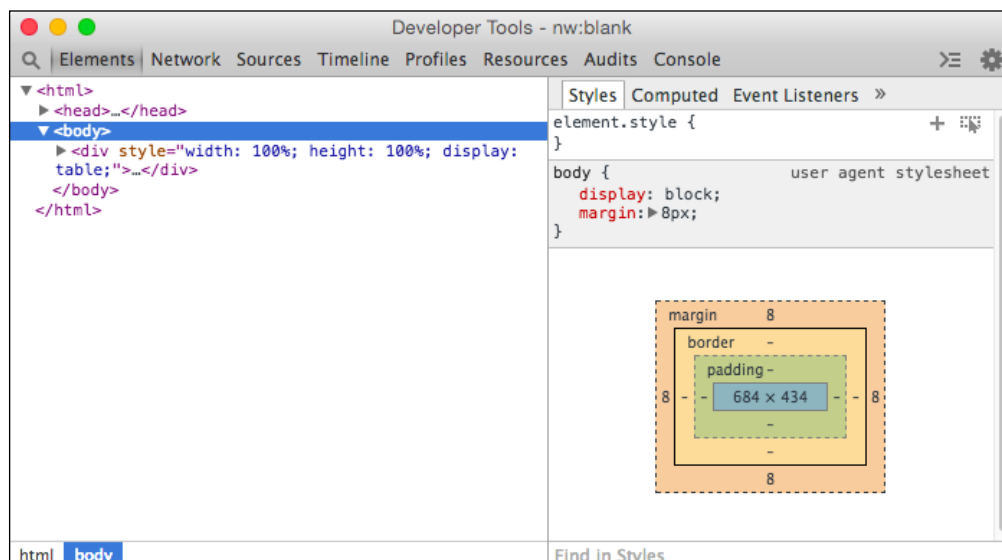
8

Let's Debug Your Application

In previous chapters, you dealt with **NW.js DevTools**. You also figured out that NW.js DevTools aren't very much different from those you're used to from developing frontend on **Google Chrome**.

NW.js DevTools are remote debugging tools where data is transferred through a socket connection from a local instanced server. Only the most relevant tools are present in NW.js DevTools as, for example, implementing the device emulator would clearly not serve any purpose.

By default, in order to open DevTools, the `window.toolbar` option in the manifest file must be set to `true`, as described in *Chapter 6, Packaging Your Application for Distribution*. Once the application is open, you'll have to click on the gear button in the top-right corner of the window's toolbar to make the DevTools window appear:



Let's examine these tools and their functions panel by panel:

- The **Elements** panel shows a real-time representation of the DOM. The user not only can easily edit the rendered HTML and CSS of the page but can also monitor JavaScript events and properties associated with each element. Using the magnification lens, you'll be able to choose a given element directly from the browser window.
- The **Network** panel contains a list of all the resources loaded by the page with a good level of detail on their time of loading and source (including methods, headers, and the response of each remote or local call).
- The **Sources** panel provides a graphical interface to the V8 debugger. This tool is pretty complex and powerful, but by simplifying, we can say that it allows you to debug issues monitoring JavaScript variables and events bringing you closer to the source of the problem, one breakpoint at a time.
- The **Timeline** panel gives you a complete overview of the timings of the application, from loading resources to JavaScript parsing and rendering.
- The **Profiles** panel allows the developer to monitor CPU and memory usage of the application, providing ways to optimize code performances.
- The **Resources** panel gives us access to the content of the many storage technologies implemented by the browser (see *Chapter 4, Data Persistence Solutions and Other Browser Web APIs*).
- The **Audits** panel is probably not as useful as it would be in a web application. However, it allows us to get a detailed overview with point-by-point improvement suggestions on the network utilization and web page performances.
- The **Console** panel is the developer's best friend as it allows real-time interaction with the DOM and the JavaScript code of the page.

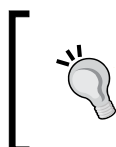
It wasn't my intention to trivialize the features of this powerful tool, but it's really not the point of the book to dive deep into all the functionalities of Chromium DevTools. If you're already familiar with it, you're good to go; otherwise, you can find a much better explanation at <https://developer.chrome.com/devtools>.

While debugging is very important to any development process, when working with a web-based, young, and bug-prone project such as NW.js, it becomes absolutely essential. That's why, NW.js provides error logging directly from the IDE (as we have seen in *Chapter 1, Meet NW.js*), **remote debugging**, and a set of **APIs** to interact with DevTools.

Remote debugging

In order to enable remote debugging, you have to add the `--remote-debugging-port=port` parameter to NW.js. You can do this from the command line / terminal:

```
$ nwjs --remote-debugging-port=9222 /path/to/your/project
```



As a reviewer of the book pointed out in NW.js v0.12.0, the remote debugging feature is not working correctly and is showing a blank page. In order to fix the issue, make sure to upgrade at least to v0.12.1.

You can also enable remote debugging directly from the **Sublime Text** NW.js builder script. To edit the build file previously created in *Chapter 1, Meet NW.js*, you have to open Sublime Text and navigate to **Preferences | Browse Packages** located relatively on the **Files** menu in Microsoft Windows or on the **Sublime Text** menu in Mac OS X.

The Packages folder will be shown. Now open the user folder, locate the `nw.js.sublime-build` file, and open it in Sublime Text. Now you can **edit the first line** of the file, as follows:

```
"cmd": ["nwjs", "--remote-debugging-port=9222",
        "${project_path}:${file_path}"]
```



Remember to change the NW.js executable name according to your operating system: `nwjs` for Mac OS X, `nw` for Linux, and `nw.exe` for Microsoft Windows.

Whether you are doing it in Sublime Text or from the terminal, once you have run the application, you can open DevTools directly from your browser at `http://localhost:9222/`.

In the page that opens, you could find one or more links. Click on the one with the name of your application and an instance of DevTools will show up so that you can start debugging the application remotely.

The DevTools API

The DevTools API is a part of the Window APIs seen in *Chapter 2, NW.js Native UI APIs*, and it provides three methods:

- `Window.isDevToolsOpen()`
- `Window.closeDevTools()`
- `Window.showDevTools([id | iframe, headless])`

The first two methods are pretty much self-explanatory, while the third one deserves to be dived deep into.

The most common use of `Window.showDevTools` is without parameters. This way, you can open DevTools at runtime without having to enable the window toolbar:

```
require('nw.gui').Window.get().showDevTools();
```

The first parameter `[id|iframe]` allows us to set the target frame for which debugging is to be enabled. Let's see an example:

```
<iframe src= "test.html" id="toDebug" ></iframe>
<script type="text/javascript">
var win = require('nw.gui').Window.get();
win.showDevTools('toDebug');
</script>
```

This way, DevTools will be enabled only on the `#toDebug` iframe. Its value can be set to the ID or the object of the iframe element.

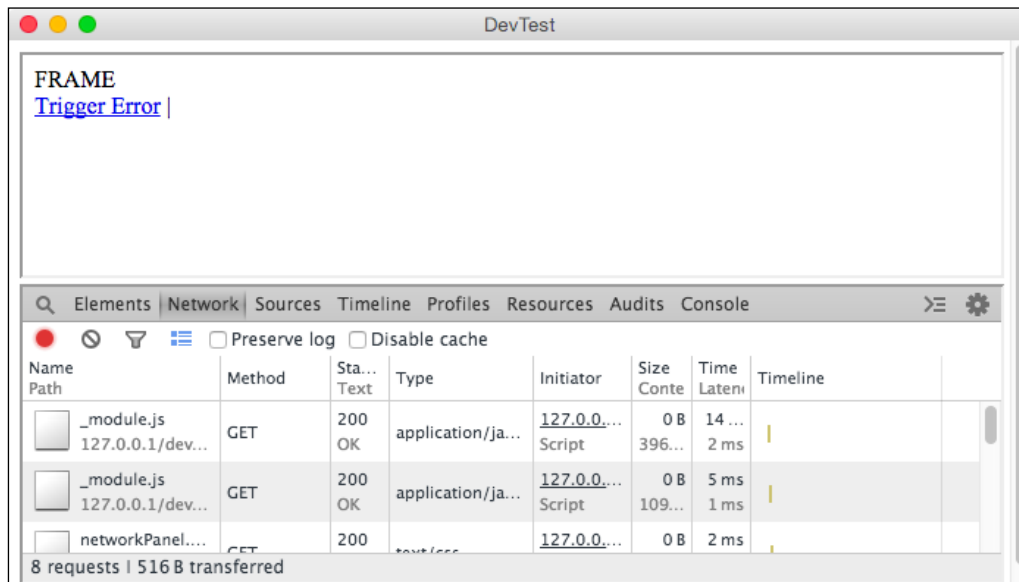
The second parameter, `headless`, comes in particularly handy when developing Web IDEs. In that case, you might want to enable the user to preview and debug its code directly inside the IDE, so you'll need to show DevTools within the IDE interface.

In order to do so, you'll have to set `headless` to `true` and load DevTools inside an iframe. Let's see an implementation example, create an example HTML file named `idePreview.html`, and then add the following code to your main file (`index.html`):

```
<iframe id="toDebug" src="idePreview.html" style="width
:100%"></iframe><br>
<iframe id="devTools" style="width:100%;"></iframe>
<script type="text/javascript">
var win = require('nw.gui').Window.get();
win.showDevTools('toDebug', true);
```

```
win.on("devtools-opened", function(url) {
  document.getElementById('devTools').src = url;
});
</script>
```

The preceding code should give the following output:



In this case, we're inspecting the `#toDebug` iframe, showing the rendered preview of an HTML file. DevTools will be opened straightaway inside the `#devTools` iframe. In order to do so, we intercept the `devtools-opened` event, which passes a URL attribute that points directly to the DevTools page.

You can even bring it a step further and use the intercepted URL to open DevTools in the default browser much as you would do for **remote debugging**.



At the time of writing, on NW.js v0.12.1, headless DevTools are not working. Hopefully, the issue should be fixed in one of the upcoming releases. As a workaround, you can temporarily add `"node-remote": "127.0.0.1"` to your manifest file. This will remove the same-origin error; however, an external DevTools window will still open and **remember to remove it from production to avoid security issues**.

Live reloading NW.js

While developing the frontend of the application, it comes in really handy to be able to reload the application interface at each change in the code. There are many approaches to **live reload**; let's see a couple of them.

If you're working on a single folder, all you need to do is leverage Node.js by adding the following code snippet directly at the end of your file:

```
<script>
var path = './',
    fs = require('fs');
fs.watch(path, function() {
  if (location) location.reload();
});
</script>
```

If you need a more complete solution by watching files recursively, you'll have to adopt a watch library, such as **Gaze**, **Gulp**, or **Chokidar**. Let's see another example, leveraging Gulp this time. First, you need to install Gulp (manually or through the manifest file) and then add the following piece of code at the end of your file:

```
<script>
var gulp = require('gulp');
gulp.task('reload', function () {
  if (location) location.reload();
});
gulp.watch('**/*', ['reload']);
</script>
```

These are only a few examples extracted from the NW.js Wiki page. You can find many more at <https://github.com/nwjs/nw.js/wiki/Livereload-nw.js-on-changes>.

Troubleshooting common issues

In *Chapter 1, Meet NW.js*, we saw how to fix the `libudev.so.0` issue, but there are a few other common issues addressed in the NW.js Wiki page:

- **Audio issues:** This might depend on using nonfree media formats, such as MP3; check *Chapter 4, Data Persistence Solutions and Others Browser Web APIs*, for more information on the subject.

- **Video issues:** Your video card might be blacklisted; in that case, run NW.js by passing the `--ignore-gpu-blacklist` attribute inside the `chromium-args` object in the manifest file. If it still does not work, it might be due to the lack of a DLL library; try copying `d3dx9_43.dll` and `d3dcompiler_43.dll` from the Chrome installer.
- **DevTools are not showing:** As stated previously, DevTools are run on a local instanced server. Proxy settings or custom host configurations might interfere with their visualization.
- **JavaScript blocks CSS animations:** JavaScript and CSS animations run inside the same thread. You can change this behavior by adding the `--enable-threaded-compositing` flag inside the `chromium-args` object in the manifest file.

Summary

There is much to add on the subject of debugging for NW.js. As stated in the introduction, NW.js is very powerful but also pretty young. Depending on the release you pick for deploying your application, you might stumble on bugs or a lack of implementations that could drive you crazy.

That's why the issues page exists (<https://github.com/nwjs/nw.js/issues>), with more than 1,700 closed issues but over 1,000 still open.

Also, to address these kinds of issues, from version 0.8, NW.js supports Crash dumps. I haven't dived deep into the argument as I find it a *bit too much* for the purpose of the book. However, feel free to learn more about it at <https://github.com/nwjs/nw.js/wiki/Crash-dump>.

We're almost done. In the last chapter, we're going to see the subsequent steps and list resources and tools for you to keep learning about NW.js.

9

Taking Your Application to the Next Level

Finally, we are at the end of this journey. Hopefully, you enjoyed the ride as much as I did. Now you have to think about the next steps to take in order to master the subject and build amazing NW.js applications. So, before leaving, I thought it would be appropriate to suggest a few tools and learning resources to simplify the process.

NW.js boilerplates

When starting a new NW.js project, it comes in really handy to have some kind of boilerplate that provides not only all the basic stuff but also coding standards, tools, and architectural hints (we've already seen the **generator-node-webkit** in *Chapter 7, Automated Packaging Tools*).

There are many technologies involved, most of which you might never have heard of, and you'll need to learn before getting serious, so *pick carefully the one that best suits your needs*.

node-webkit-hipster-seed

Download link: <https://github.com/Anonyfox/node-webkit-hipster-seed>

This is one of the most popular NW.js boilerplates. Between its ingredients, we can find *CoffeeScript*, *Less*, *Jade*, *Angular.js*, and *Twitter Bootstrap*. We can also find *Brunch* for code compilation, *Karma* for testing, and *grunt-node-webkit-builder* for building NW.js packages.

angular-desktop-app

Download link: <https://github.com/jgrenon/angular-desktop-app>

This is a simple application skeleton to create desktop applications using NW.js and *Angular.js*. It uses *Require.js* for *Angular.js* dependencies, *Bower* for client-side libraries, *Twitter Bootstrap* for styles, and *grunt-node-webkit-builder* for building.

node-webkit-tomster-seed

Download link: <https://github.com/Kerrick/node-webkit-tomster-seed>

This is a fork of *node-webkit-hipster-seed* that uses *Ember.js* instead of *Angular.js*.

node-webkit-boilerplate

Download link: <https://github.com/brandonjpierce/node-webkit-boilerplate>

This is a simple boilerplate that uses *Vue* as its MVVM and *Gulp* and *Browserify* for testing. It's the only one that doesn't provide a NW.js builder (it might be a good thing or a bad thing; it's up to you to decide).

nw-boilerplate

Download link: <https://github.com/szwacz/nw-boilerplate>

This is another boilerplate for NW.js that provides a custom *Gulp* task for building. A pretty cool thing about this boilerplate is that it comes with *es6-module-transpiler*, which allows you to use the ES6 draft specification module syntax. The resulting files will then be translated into AMD modules. It also comes with *Jasmine* for testing.

Development ideas

It would be simplistic to think that you all share the same goals, so I thought of providing you with a few case scenarios you can take inspiration from:

- **Game development:** If you're already into browser game development, you are probably set to go; otherwise, you should know that there are plenty of JavaScript frameworks that can simplify the task. The first one that I can think of is *Phaser*, an open source desktop and mobile HTML5 game framework. With support for *WebGL*, *canvas*, *physics*, and *tilemaps*, it will make game development a breeze (<http://phaser.io/>).

- **System tool development:** If you are planning to release a system tool, having a pretty good knowledge of Node.js/io.js is mandatory. That's something you can learn from books, online courses, and a lot of experience in the field, so there's not much I can tell you about it. However, once you've acquired the needed skills, you'll have to think about the look and feel of the application, so I'd love to introduce you to **pawnee**, an open source tray bar utility designed to help you manage your local Apache installation. What really impressed me about the application is its beautiful GUI, which pops up from the tray bar. You should definitely take a look at its source code (<https://github.com/johansatge/pawnee>).
- **IDEs and development tools:** These kinds of applications leverage both Node.js and Chromium features. IDEs are usually very complex, but fortunately, you can start out with **Ace**, an advanced, standalone code editor written in JavaScript. We are talking about the same editor that powers up GitHub; it is extensible, and is full of features (<http://ace.c9.io/>). Of late, you can find more and more open source tools for developers, and I hope the trend will keep going that way. To have that many choices is amazing; however, sometimes, I feel that having 10 developers working on a single open source project would be much more effective than having 10 different projects that are all incomplete. My point is that before starting a brand-new project, it would be better to check whether something similar already exists and figure out whether you can possibly improve it. That said, I invite you to take a look at some cool NW.js development tools that you should absolutely check out:
 - **Koala**, a GUI application for Less, Sass, Compass, and CoffeeScript compilation (<http://koala-app.com/>)
 - **Fenix**, a simple static desktop web server (<http://fenixwebserver.com/>)
 - **Gisto**, a cross-platform gist snippet manager (<http://www.gistoapp.com/>)
- **Markdown editor:** From my very humble point of view, there are already too many markdown editors out there, so if you really have to, think about participating in **Haroopad** development (<https://github.com/rhiokim/haroopad>).

- **Nonconventional uses, virtual reality, and robotics:** Alright, this is a kind of borderline; however... wouldn't it be cool to easily create GUIs for those ugly CLIs used in robotics? Here are a couple of tutorials that might fit your needs:
 - **Arduino, Johnny-Five, and Nw.js:** <https://github.com/rwaldron/johnny-five/wiki/Getting-started-with-Johnny-Five-and-Node-Webkit>
 - **Oculus Rift Hello World with NW.js:** <http://tyrovr.com/2014/06/01/nw-ovr-sdk-tutorial.html>

Resources and tutorials

- **Creating peer-to-peer NW.js applications:** <http://laike9m.com/blog/a-tutorial-on-using-peerjs-in-node-webkit-app,57/>
- **Getting started with NW.js apps:** <http://thejackalofjavascript.com/getting-started-with-node-webkit-apps/>
- **HTML5DevConf – WebApp to DesktopApp with NW.js:** <https://www.youtube.com/watch?v=d2tYH7vXMUM>
- **Run/debug NW.js with JetBrains WebStorm IDE:** <https://www.jetbrains.com/webstorm/help/run-debug-configuration-node-webkit.html>
- **COLT – NW.js live coding tool:** <http://codeorchestra.com/>
- **NW.js and Angular.js tutorial:** <http://thejackalofjavascript.com/node-webkit-and-angularjs-a-moviestub-app/>
- **Creating a Markdown editor tutorial; sorry for that... (just kidding):** <http://duco.cc/node-webkit-tutorial-creating-a-markdown-editor/>

Summary

In this last chapter, I gave you a general smattering of tools, resources, and tutorials revolving around NW.js.

The point is that the community is the real strength of this library. Many devoted and enthusiastic developers find some spare time to improve NW.js between job, study, and family.

In contrast to other proprietary solutions, NW.js depends on you – on your ability to run tests, find and share bugs, and implement bug fixes or new features. It's a complex ecosystem where each user can do its part.

So, I thought that the best way to end the book was to thank you and provide you with a couple of channels to get in touch with the amazing NW.js community:

- **NW.js Gitter channel:** <https://gitter.im/nwjs/nw.js>
- **NW.js Google group:** <https://groups.google.com/forum/#!forum/nwjs-general>

Index

Symbols

"Hello World" application

- running, on Linux 13, 14
- running, on Mac OS 13
- running, on Microsoft Windows 12
- running, on Sublime Text 2 12
- writing 10-12

A

absolute path 56

Ace

- URL 161

angular-desktop-app

- about 160
- URL 160

AngularJS 50

App API

- about 17
- application data folder path,
 - accessing 19, 20
- applications, closing 21, 22
- file, opening 18, 19
- manifest file data, accessing 20
- other APIs 23
- system-wide hotkeys, registering 22, 23

Application Binary Interface (ABI) 60

application launchers 131

application layer, TODO list application

- export feature, implementing 103-105
- implementing 96-98
- new task, adding 98-101
- sync feature, implementing 103-105
- tasks, loading 102, 103

array casting 59

Audits panel, NW.js DevTools 152

automated packaging tools 135

B

Binary Large Objects (BLOBs)

- about 65
- and XMLHttpRequest 79, 80

Blink 3

boilerplates

- about 159
- angular-desktop-app 160
- node-webkit-boilerplate 160
- node-webkit-hipster-seed 159
- node-webkit-tomster-seed 160
- nw-boilerplate 160

Browser Web APIs 63

built-in Mac menu 40

C

Cellist 5

Chokidar 156

Chromium 63

Chromium DevTools

- URL 152

client-server applications 49

Clipboard API

- about 17
- system clipboard, accessing 47

components, NW.js

- Node.js 3
- WebKit 3

Console panel, NW.js DevTools 152

context issues

- about 57, 58
- reference link 59

contextual menu

- about 37
- handling 38-40

CouchDB

- about 95, 104
- URL 79

Crash dump feature 23

cross-origin access 23

cursor 78

customization options, manifest file

- additional_trust_anchors 122
- chromium-args 122
- dom_storage_quota 123
- inject-js-end 122
- inject-js-start 122
- js-flags 122
- main 120
- name 120
- nodejs 120
- node-main 120
- node-remote 122
- single-instance 120
- snapshot 123
- user-agent 122
- version 120
- webkit 122
- window 120-122

D

data persistence, solutions

- about 64
- IndexedDB 72-79
- Web SQL database 68-72
- web storage 65-67

development ideas

- development tools 161
- game development 160
- IDEs 161
- markdown editor 161
- system tool development 161

development tools

- using 9, 10

Devtools window 16

Dexie

- URL 79

E

EditorConfig

- URL 147

Elements panel, NW.js DevTools 152

Ember.js 50

error handling 75

event-driven runtime environment 49

EventEmitter, Node.js 16

Express 50

F

Fedora 8

Fenix

- about 161
- URL 161

file dialogs

- about 17, 41
- default path, suggesting 43
- directory, opening 43
- files, opening 41, 42
- files, opening through file dragging 44
- files, saving 41-43
- filtering, by file type 43
- multiple files, opening 43

flex 92

G

Game Dev Tycoon 6

Gaze 156

generator-node-webkit

- example 146-149
- installing 145
- URL 145

Gisto

- about 161
- URL 161

global object 51, 52

global root variable 52

Google Chrome 151

grunt-node-webkit-builder

- about 137, 138
- example 144, 145
- URL 137

Grunt plugin version 138

Gulp 138, 156

H

Haroopad

about 161
URL 161

Homebrew Cask 7

HTML5 APIs 44, 63

HTML5 Application Cache 64

I

iConvert Icons

URL 125

img2icns

URL 125

IndexedDB 72-79

Inno Setup

URL 124
using 124

installation, NW.js

about 6
on Linux 8, 9
on Mac OS X 6, 7
on Microsoft Windows 7

Intel® XDK 6

internal modules 59

io.js

about 2
URL 9

J

jQuery

about 42
reference link 42
using 50, 88

jQuery 2.1.x

about 89
URL 89

JSHint

URL 147

K

Kiosk mode, Window API 29, 30

Koala

about 161
URL 161

L

license, NW.js applications

about 134
URL 134

Linux

"Hello World" application, running 13, 14
file type associations, adding 131, 132
icon, adding 131, 132
NW.js applications, packaging 130, 131
NW.js, installing 8, 9

live reload

about 156
reference link 156

M

Mac OS X

"Hello World" application, running 13
file extension, associating with
application 127, 128
NW.js applications, packaging 125-127
NW.js, installing 6, 7

main module

using 54, 55

manifest file

customizing 120-123
fields 123

markdown module 11

media files

handling 80, 81

Menu API

about 17, 37
contextual menu, handling 37-40
tray icon menu, handling 38
window menu, handling 37-41

Microsoft Windows

"Hello World" application, running 12
file type association, registering 130
NW.js applications, packaging 128-130
NW.js, installing 7

modules, Node.js

about 59
internal modules 59
third-party modules, with C/C++
add-ons 60
third-party modules, written in
JavaScript 60

N

native desktop application 15

Native UI APIs

about 15, 16, 120

App 17

Clipboard 17

file dialogs 17

Menu 17

reference link 17

Screen 17

Shell 17

Tray 17

Window 17

NativeUI layer, TODO list application

Context menu, implementing 111

implementing 105, 106

Options window, implementing 112-116

Window menu, implementing 106-111

window position, restoring 111

Network panel, NW.js DevTools 152

Node frames

and Normal Frames 81-83

Node.js

about 2, 3, 15

global object 51, 52

process object 51, 52

URL 9

Node-Webkit. See NW.js

node-webkit-boilerplate

about 160

URL 160

node-webkit-builder

about 137

examples 138-143

installing 138

URL 137

node-webkit-hipster-seed

about 159

URL 159

node-webkit-tomster-seed

about 160

URL 160

Normal Frames

and Node frames 81-83

npm 59

nw-boilerplate

about 160

URL 160

nwglobal module

about 58

URL 58

nw-gyp

using 60

NW.js

about 1, 2

components 3, 4

downloading 6

drawbacks 4

features 4

installing 6

installing, on Linux 8, 9

installing, on Mac OS X 6, 7

installing, on Microsoft Windows 7

live reloading 156

resources 162

URL, for downloading 6

usage scenarios 5

NW.js applications

Cellist 5

file extension, associating for

Mac OS X 127, 128

file type association, registering on

Microsoft Windows 130

file type associations, adding on

Linux 131, 132

Game Dev Tycoon 6

icon, adding on Linux 131, 132

Intel® XDK 6

licensing 134

packaging, for Linux 130, 131

packaging, for Mac OS X 125-127

packaging, for Microsoft Windows 128-130

Wunderlist, for Windows 7 5

NW.js DevTools

about 151

API 154, 155

Audits panel 152

Console panel 152

Elements panel 152

Network panel 152

Profiles panel 152

Resources panel 152
Sources panel 152
Timeline panel 152

P

Package Control
URL 10
package.json manifest file 11
packaging procedure
general logic 123, 124
path module 56
paths
handling 55, 56
pawnee
about 161
URL 161
Phaser
about 160
URL 160
plain JavaScript 49
PouchDB
about 95
URL 79
PouchDB 3.2.x
URL 89
process object 51, 52
Profiles panel, NW.js DevTools 152

R

remote debugging
about 152, 155
enabling 153
RequireJS
URL 59
ResourceHacker
URL 129
Resources panel, NW.js DevTools 152
routing 50, 51

S

Screen API 17, 36, 37
Shell API
about 17, 48
platform-dependent desktop functions 48

Shortcut API 22
source code
securing 132, 133
Sources panel, NW.js DevTools 152
SQL language 68
StoreDB
URL 66
Sublime Text 153
Sublime Text 2
"Hello World" application, running 12
about 9
URL 9
Swig template engine 89
Swig templates
about 51
URL 100
system tray 44
system-wide hotkeys
registering 22, 23

T

taskbar icon, Window API 32, 33
templating 50, 51
third-party modules
with C/C++ add-ons 60
written in JavaScript 60
Timeline panel, NW.js DevTools 152
ToDo list application
about 88
application layer 96-98
application logic 94-96
closing 116, 117
folder structure, creating 88, 89
HTML5 skeleton, creating 93, 94
NativeUI layer, implementing 105, 106
opening, smoothly 117
style sheets, creating 90-92
Tray API
about 17, 44
application, hiding 44-47
tray icon menu 38
Tray object 45
troubleshooting, common issues
audio issues 156
CSS animations, blocking 157

- DevTools visualization issue 157
- video issues 157
- two-way data replication** 95

U

- Ubuntu** 8

W

- W3C specifications**

- URL 79

- Web2Executable**

- about 135, 136

- building procedure 136, 137

- URL 135

- web application frameworks** 50

- web app runtime** 2

- WebKit** 3, 15

- Web Notifications API** 83, 84

- Web SQL Database**

- about 68-71

- URL 72

- web storage**

- localStorage 64

- sessionStorage 64

- Window API**

- about 16, 17, 24

- drag regions 31, 32

- events 35

- frameless windows 31, 32

- fullscreen windows 29, 30

- Kiosk mode 29, 30

- other Window APIs 34, 35

- taskbar icon 32, 33

- window object, instantiating 24-26

- window position, setting 26, 27

- windows, closing 33, 34

- window size, setting 26, 27

- window status, modifying 28, 29

- Window menu**

- about 37

- handling 40, 41

- window object** 53, 54

- Windows Node.js module**

- URL 124

- using 124

- Wine**

- URL 142

- Wunderlist**

- for Windows 7 5

X

- X11** 131

- XMLHttpRequest**

- and BLOBs 79, 80



Thank you for buying NW.js Essentials

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

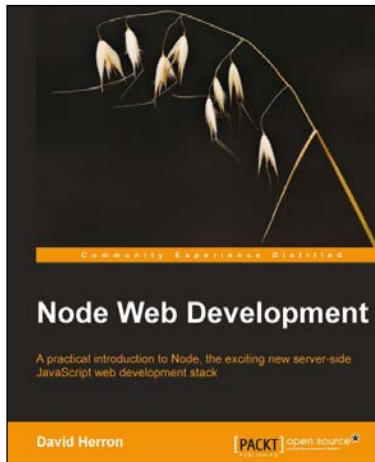
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



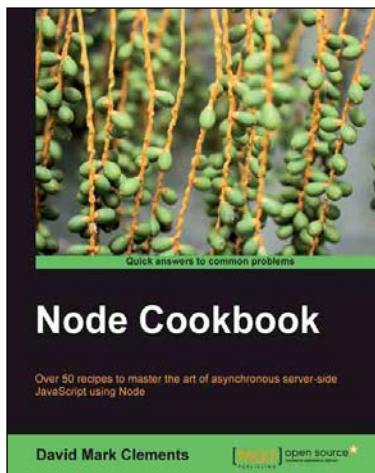
Node Web Development

ISBN: 978-1-84951-514-6

Paperback: 172 pages

A practical introduction to Node, the exciting new server-side JavaScript web development stack

1. Go from nothing to a database-backed web application in no time at all.
2. Get started quickly with Node and discover that JavaScript is not just for browsers anymore.
3. An introduction to server-side JavaScript with Node, the Connect and Express frameworks, and using SQL or MongoDB database back-end.



Node Cookbook

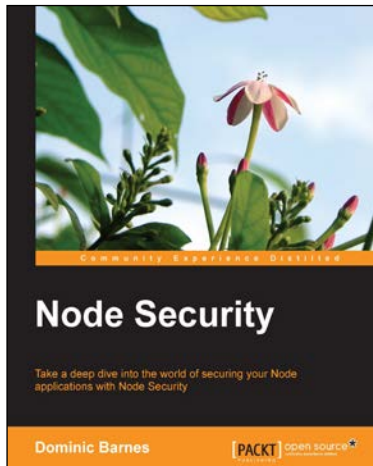
ISBN: 978-1-84951-718-8

Paperback: 342 pages

Over 50 recipes to master the art of asynchronous server-side JavaScript using Node

1. Packed with practical recipes taking you from the basics to extending Node with your own modules.
2. Create your own web server to see Node's features in action.
3. Work with JSON, XML, web sockets, and make the most of asynchronous programming.

Please check www.PacktPub.com for information on our titles



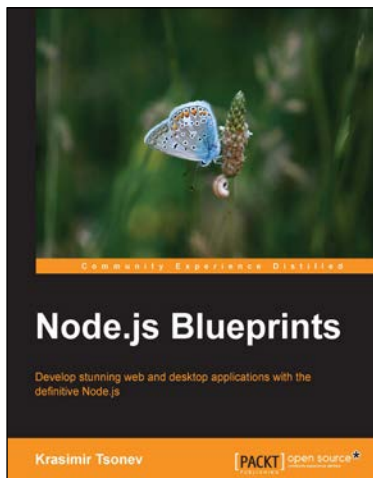
Node Security

ISBN: 978-1-78328-149-7

Paperback: 94 pages

Take a deep dive into the world of securing your Node applications with Node Security

1. Examine security features and vulnerabilities within JavaScript.
2. Explore the Node platform, including the event-loop and core modules.
3. Solve common security problems with available npm modules.



Node.js Blueprints

ISBN: 978-1-78328-733-8

Paperback: 268 pages

Develop stunning web and desktop applications with the definitive Node.js

1. Utilize libraries and frameworks to develop real-world applications using Node.js.
2. Explore Node.js compatibility with AngularJS, Socket.io, BackboneJS, EmberJS, and GruntJS.
3. Step-by-step tutorials that will help you to utilize the enormous capabilities of Node.js.

Please check www.PacktPub.com for information on our titles